R2PL 2005—Proceedings of the First International Workshop on Reengineering Towards Product Lines

Bas Graaf
Liam O'Brien
Rafael Capilla

March 2006

SPECIAL REPORT
CMU/SEI-2006-SR-002

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

R2PL 2005—Proceedings of the First International Workshop on Reengineering Towards Product Lines

Bas Graaf
Liam O’Brien
Rafael Capilla

March 2006

CMU/SEI-2006-SR-002

Product Line Practice Initiative

Table of Contents

Executive Summary .... **vii**

Abstract .... **ix**

1 Background **1**

2 Workshop Organization **3**

3 Invited Talk: Consolidating Software Variants into Software Product Lines—A Research Outline **5**

4 Quality-Driven Conformance Checking in Product Line Architectures **7**

4.1 Introduction 7

4.2 Challenges and Related Work 9

4.3 Variation Point Concepts Usage 10

4.4 Conclusion and Open Issues 13

4.5 References 14

5 Identification of Variation Points Using Dynamic Analysis **17**

5.1 Introduction 17

5.2 Related Work 18

5.3 Tracing and Variation Points 18

5.4 Approach 20

5.4.1 Running example: Pacman 20

5.4.2 Dynamic analysis using aspects 20

5.4.3 Determining variation points 20

5.5 Preliminary Results 21

5.5.1 Generating traces 22

5.5.2 Branching behavior 23

5.6 Discussion and Future Work 24

5.7 Acknowledgements 25

5.8 References 25

6 Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product Line Migration ... 27

6.1 Introduction ... 27

6.2 Approach ... 29

6.2.1 Terminology ... 29

6.2.2 Factors Affecting Reusability ... 29

6.2.3 Metrics for Measuring Costs, Usefulness and Quality ... 31

6.3 Process ... 33

6.4 Related work ... 35

6.5 Conclusion and Future Work ... 35

6.6 References ... 36

7 Analyzing the Product Line Adequacy of Existing Components ... 37

7.1 Introduction ... 37

7.2 Context ... 38

7.3 Approach ... 38

7.3.1 PuLSE™-DSSA ... 38

7.3.2 ADORE™ ... 39

7.4 Analysis of Component Adequacy ... 41

7.4.1 Static Architecture Evaluation ... 41

7.4.2 Variability Analysis ... 43

7.4.3 Clone Detection ... 43

7.4.4 Metric Hotspots ... 44

7.4.5 Naming and Decomposition Analysis ... 44

7.4.6 Code Comments ... 44

7.5 Summary ... 44

7.6 References ... 45

8 Mining Existing Software Product Line Artifacts using Polymorphic Dependency Relations ... 47

8.1 Introduction ... 47

8.2 Semantic Annotation of Software Artifacts ... 48

8.3 Defining the Mining Context ... 49

8.4 Conclusions and Future Research ... 52

8.5 Acknowledgements ... 52

8.6 References ... 52

9 Workshop Outcomes ... 55

References ... 57

List of Figures

Figure 4-1: Example product line architecture adapted from [7] 11

Figure 5-1: Forks and merges in an execution trace.. 19

Figure 5-2: A fork and its context in the trace. 22

Figure 5-3: A merge and its context in the trace.. 24

Figure 6-1: Process model for component extraction.. 29

Figure 6-2: Factors affecting reusability [4] 30

Figure 6-3: Associating OO metrics with factors affecting reusability.................... 32

Figure 6-4: Different steps for component extraction 33

Figure 7-1: Overview PuLSE-DSSA and ADORE 40

Figure 7-2: Component Internal Layers.. 42

Figure 7-3: Frame Hierarchy 42

Figure 8-1: Annotation Context UML Profile... 49

Figure 8-2: Mining Context as a UML Profile.. 50

Figure 8-3: Attribute Association Rules for Mapping Heterogeneous Semantic Values 51

List of Tables

Table 6-1: Reusable Aspects of Software Projects 27

Table 6-2: Definition of Metrics 31

Table 6-3: Metrics with Lower and Upper Bound 33

Executive Summary

This report contains the proceedings from the First International Workshop on Reengineering Towards Product Lines (R2PL) 2005, which was held on November 10th, 2005 in Pittsburgh, Pennsylvania, USA and colocated with the Working Conference on Reverse Engineering (WCRE) 2005 and WICSA 2005—the Working Institute of Electrical and Electronics Engineers/International Federation for Information Processing (IEEE/IFIP) Conference on Software Architecture. This report consists of an overview of an invited presentation, a set of position papers, and details of the workshop's outcomes.

Abstract

This report contains the proceedings from the First International Workshop on Reengineering Towards Product Lines (R2PL) 2005, which was held on November 10th, 2005 in Pittsburgh, Pennsylvania, USA and colocated with the Working Conference on Reverse Engineering (WCRE) 2005 and WICSA 2005—the Working Institute of Electrical and Electronics Engineers/International Federation for Information Processing (IEEE/IFIP) Conference on Software Architecture. This report consists of an overview of an invited presentation, a set of position papers, and details of the workshop's outcomes.

1 Background

Today, software-intensive systems are developed more and more using product line approaches. These approaches require the definition of a product line architecture that implicitly or explicitly specifies some degree of variability. This variability is used to instantiate concrete software product instances. A product line approach not only implies reuse of architecture-level design knowledge, it also facilitates reuse of implementation-level artifacts, such as source code and executable components. The use of software product lines can reduce the cost of developing new products significantly.

In practice, software products are usually not developed from scratch. Software product lines are typically introduced following an evolutionary approach. First, a product line architecture is defined based on an initial set of products. Then, the scope of the product line is gradually extended by incorporating more existing and new products. Before a product line is extended, its suitability for incorporating more products needs to be evaluated, as well as the extent to which the new and currently included products conform to the product line architecture.

For companies adopting a product line approach for their software development, the problem remains of how to reuse as much as possible of the existing legacy development artifacts. Reuse can be applied to the definition and implementation of a product line architecture and to the specifications and implementation of concrete product instances based on (legacy) software development artifacts. In this workshop [Graaf 05, R2PL 05], we discuss the use of reverse engineering and reengineering technology to solve the problems described above.

The papers included in this report appear exactly as they did in the original presentations; they have not been edited further (aside from adjusting their section numbers for the new layout in this report).

2 Workshop Organization

Organizers

Bas Graaf
Delft University of Technology
Delft, The Netherlands
b.s.graaf@ewi.tudelft.nl

Liam O'Brien
Software Engineering Institute
Pittsburgh, PA, USA
lob@sei.cmu.edu

Rafael Capilla
Universidad Rey Juan Carlos
Madrid, Spain
rafael.capilla@urjc.es

Program Committee

Liam O'Brien
Software Engineering Institute

Rafael Capilla
Universidad Rey Juan Carlos

Arie van Deursen
Delft University of Technology

Gerald C. Gannod
Arizona State University

Bas Graaf
Delft University of Technology

3 Invited Talk: Consolidating Software Variants into Software Product Lines—A Research Outline

Rainer Koschke
University of Bremen
Germany

Abstract

Software product lines often arise from a set of variants of a common code basis that have been individually adapted to a particular requirement variability. This ad-hoc and unplanned approach causes serious maintenance problems. Migrating such variants into an organized software product line promises better maintainability.

In this talk, I shall outline our 3-year research program aiming at consolidating software variants into software product lines. We are tackling the problem both at the source code level and architectural level. We are adapting and extending techniques, such as clone detection, feature location, protocol recovery, and reflexion-based reconstruction that we have so far applied only to individual systems.

4 Quality-Driven Conformance Checking in Product Line Architectures

Femi G. Olumofin
Vojislav B. Mišić
University of Manitoba, Winnipeg,
Manitoba, Canada

Abstract

Software product lines are often developed through reengineering existing products and legacy applications. In such cases it is not uncommon for the behavioural and quality characteristics of individual product architectures to be inconsistent with those of the common architecture. Successful development of product lines dictate that those inconsistencies be resolved. The resolution process involves bringing the product architecture into structural, semantic and quality attribute-related congruence with the common architecture. Additional steps must be taken to ensure their continued conformance in order to facilitate subsequent maintenance and evolution activities. In this paper, we describe a simple design-time technique that aims to ensure that quality attribute responses of individual product architectures are in conformance with those of the common architecture. The technique is based on the concept of variation points.

4.1 Introduction

For more than a decade, software architecture has been steadily gaining importance as the most effective vehicle for the development of complex software intensive systems. Architecture-based design offers unmatched flexibility and allows crucial insights to be obtained very early in the design cycle. Architectural abstraction avoids complex code level details while making component structures and interrelationships explicit. In this manner, the use of architecture facilitates human understanding of the system as well as reasoning about quality characteristics and attributes. It should come as no surprise, then, that the reengineering of existing systems and legacy applications—recovering their structure in order to develop new, functionally equivalent but improved systems—often focuses on recovering or reconstructing the architecture in the form of a product. Most such efforts are motivated by changes in quality attributes, such as extendibility and maintainability, rather than by the need for functional changes and enhancements [3, 10]. For example, consider a system that has undergone several maintenance cycles which included functionality enhancements. While the system itself may be in working order, the documentation complexity and, possibly, inconsistency make further maintenance difficult. The first thought would be to leave the system as it is and reconstruct the documentation only; but a better way is to disregard the documentation and recover the system architecture from the system itself. Oftentimes, architecture recovery is the first step towards reengineering the entire system.

All of the aforementioned advantages are even more important in the case of software product families or product lines [5]: sets of related yet distinct software intensive systems developed from the same base architecture. In the product line approach, requirements or features common to all the products are used as the basis for the so-called *core architecture*, or CA. Requirements which are specific to some of the products only, but not all of them, are represented as *variation points* in the CA. (It is common to refer to the two sets of requirements as commonality, or commonalities, and variability, respectively.) Individual products are then developed to address the specific sets of requirements. In one approach, individual products are directly developed from the CA by replacing the variation points with product-specific component instances, called *variants*. This approach is often used in simpler cases—i.e., when the number of individual products and/or variation points is not high.

In an alternative approach, the CA is used to instantiate a number of separate *product architectures* or PAs, which correspond to individual products. The PA is created from the CA by exercising the built-in variation points. The actual products are then developed from the corresponding PAs. This *dual form* of representation of the architecture (i.e., CA and PA) is typical of the software product lines [5, 6].

Yet more problems arise when the product line development path involves the reuse of existing products. In most cases, existing products and legacy systems were built with little care (or none at all) for consistency and quality, thus encumbering the identification of commonalities and variability required for the product line approach. Once identified and specified, the CA and the individual PAs may differ significantly, in particular with regard to consistency and prioritization of quality attributes. Any inconsistencies and differences in the architectures recovered from the existing system must be resolved in the product line architectures—successful development of the reengineered system is contingent upon the design of both CA and PAs being quality attribute-driven and conflict-free.

In this paper, we present a design-time technique for maintaining conformance between the reengineered and evolving CA and individual product architectures. The technique is based on the concept of variation points, which are exploited in a systematic fashion in order to constrain the individual PAs to be consistent with the CA. While the approach described is particularly suited to reengineering product lines, its generality makes it also applicable for validation of product line architectures developed 'from scratch' as well as those developed using the revolutionary approach [2]. The paper is organized as follows. In Section 4.2, we briefly describe the challenges of ensuring quality conformance between the CA and the PAs, and discuss some earlier work that touches this issue. Section 4.3 introduces our technique based on variation points, together with a small example that illustrates the use of the technique. Finally, Section 4.4 summarizes the paper and highlights some open issues for further work.

4.2 Challenges and Related Work

As noted above, the product line architecture consists of a core architecture (CA) which is used as the basis for developing a number of individual product architectures (PAs). The CA is necessarily underspecified, while the individual PAs must be fully specified since the actual products will be derived from them. However, the set of quality attributes for a given PA may significantly differ from that of the underlying CA, and even priorities of different attributes may differ. To consider the interplay between the quality attributes of individual PAs and those of the CA, we need to start by considering the CA. The quality attribute goals in the CA are addressed through the so-called sensitivity and tradeoff points [1, 4]. A sensitivity point is an area of the architecture which determines the responses of at least one quality attributes. A tradeoff point is an area of the architecture which determines the responses of two or more quality attributes, usually in opposing ways. (Note that each tradeoff point is a sensitivity point by default.)

The problem lies, of course, in that the individual PAs have quality attributes and priorities of their own. Satisfying those attributes may cause conflict with the decisions made in the CA, thus compromising the quality attributes that should be common to both the CA and all PAs. Namely, the changes that fully specify an underspecified CA, and thus instantiate the particular PA, are made in an area with a variation point—the requirement specific to the PA but not present in the CA itself. If the variation point overlaps with a sensitivity point of the original CA, the corresponding quality attribute may be affected. If the variation point overlaps with a tradeoff point, several of the original attributes will be affected. Now, each of the individual PAs instantiates a particular variation point from the underlying CA in its own fashion. As a result, conformance checking between the CA and individual PAs is a complex process, and the problem is not made any easier by the fact that there may be quite a few PAs derived from a single CA.

Several authors have identified this problem in the context of architecture reengineering. In most cases, such reengineering is based on updating the ‘as designed’ architecture of a system from the ‘as-built’ architecture reconstructed by reverse engineering. Once the architectural description of the existing system is accurately specified, it can be modified in order to fulfill the emergent quality goals of the new target system.

Bengtsson and Bosch present an iterative, scenario-based reengineering method for transforming software architectures to provide desired quality attributes responses [3].

QADSAR [13] is a quality attributes scenario driven reverse engineering method for architectures of existing systems, whose tool support is the ARMIN. The goal of a QADSAR reconstruction is to provide architectural description and information on architectural drivers to enable qualitative architectural analysis.

Stoermer *et al.* [12] provides a codification of six practice patterns for architectural reverse engineering. These patterns are described with a name, context of application, concise statement of problem in the context, an example illustration in an industrial context, and the expected solution/delivery from applying the pattern. The paper also describes some common approaches to reverse engineering, including tool supported approaches. The suitability of different approaches (and the accompanying tools) for use in the practice patterns is also discussed. The result of the analysis revealed the lack of adequate coverage for the practice pattern by the existing approaches.

Finally, Tahvildari *et al.* [14] proposed a quality-driven software reengineering framework similar to that of Bengtsson and Bosch [3]. This framework is based on the use of desirable target-system qualities to define and guide the reengineering. According to the Stoermer's practice pattern catalogue [12], this framework may be categorized into the quality attribute changes practice pattern. In this pattern, legacy systems are reengineered to improve some desired quality attributes responses, such as performance or maintainability.

4.3 Variation Point Concepts Usage

In order to ensure quality congruence between the common architecture and individual product architectures, both the existing and the emerging ones, we make use of the concept of variation points. Variation points are architectural placeholders for augmenting the CA with behavioural extensions. They are instantiated as concrete variants in individual product architectures. The sensitivity points are those architectural decisions that affect one or more quality goals [8]. For example, the encryption of sensitive message exchange between two components may improve the security quality of a software-intensive system. The architectural decision to introduce cryptographic components between the two communicating components is a sensitivity point intended to implement security insofar as message exchange between the two components is concerned.

Architectural decisions made in the process of defining the CA, and subsequently found to be sensitivity points to one or more quality attributes, continue to remain valid for individual product architectures. A possible exception would be the case in which the creation of a PA involves the addition of component variants to those parts of the architecture which interact with the sensitivity points. In the example given above, consider adding a third component to periodically receive exception messages from both components. If such notification messages to this third component are not similarly encrypted, the security of the system may be jeopardized.

An area of the architecture, which is a sensitivity point and which contains at least one variation point, will be referred to as an *evolvability point*. Such variation/evolvability points deserve special treatment, as they have the potential to alter (and, possibly, damage) the quality of the architecture(s). In order to defuse that potential, each evolvability point in the CA is accompanied by suitable guidelines to constrain or guide subsequent PA design decisions and conformance checking. Thus, the developers are warned against making design decisions in a PA that could invalidate the quality goals already identified in the CA.

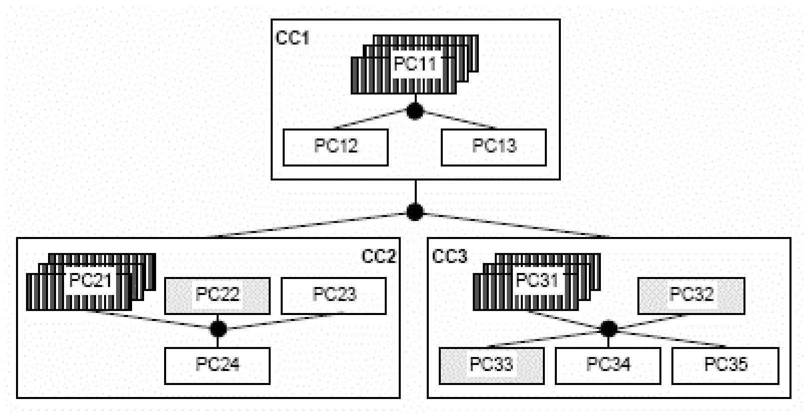


Figure 4-1: Example product line architecture adapted from [7]

(Unshaded boxes represent mandatory components; vertically striped boxes represent alternative components; shaded boxes represent optional components.)

As the CA changes, the evolvability constraints (or quality attributes conformance constraints) are updated accordingly to guide future design of the PAs. The evolvability points also help simplify maintenance because the architects would be rightly guided to those critical design decisions that control quality attribute responses.

As an example, consider the architecture shown in Figure 4-1, which is made up of three complex (or composite) components CC1, CC2, and CC3. Each of these components is in turn made up of a number of primitive components. In the product line approach, those primitive components can be identified as mandatory (or common), optional, or alternative. Mandatory components, by definition, are fully specified in the CA and are always present in any PA. Optional components are underspecified as variation points in the CA; they can become fully specified as components (or variants) in a given PA, or they will not be present at all. Finally, alternative components are underspecified as variation points in the CA but must become fully specified into some component (or variant) in the PA.

In the definition of this architecture, design decisions that interact with one or more quality attributes (i.e., sensitivity points) are assumed to be located in some of the components. Let’s assume that performance and availability are the two quality attribute goals of the highest priority. We shall consider two scenarios in relation to the architecture illustrated in Figure 4-1: in the first scenario, the architecture is taken to be a product architecture (PA), while in the second, it is taken to be the core architecture (CA).

Scenario 1: architecture is a PA

If the architecture in Figure 4-1 is a product architecture, then the shaded and unshaded boxes are fully specified architectural components (i.e., primitive components). In this scenario, we will consider two possibilities concerning the nature of the sensitivity points.

In one case, let the sensitivity points be located in the mandatory components whose design decisions are preset in the

CA. For example, the sensitivity interacting with performance is localized in PC23, while that of availability is localized in PC24. Since both sensitivity points are localized in mandatory components, each individual PA inherits those sensitivity points intact. With them, performance and availability qualities are inherited from the CA. As a result, the availability and performance quality will always be met in this PA. In fact, every product built from that CA is guaranteed to provide the preset quality responses for performance and availability.

Alternatively, one or both quality attributes may be localized in an optional or alternative component. Let us assume that the performance quality of this PA is determined through the appropriate design decisions of CC2. Further, assume those design decisions are jointly localized in components PC24 (mandatory) and PC21 (alternative). The design decisions of PC24 are determined during the CA definition, while those of PC21 are determined in this particular PA definition. If the correct guarantees for performance are provided through PC24, but not through PC21, the desired performance response may not be guaranteed. To avoid this, the PA must correctly specialize PC21 from its variation point definition in the CA; to this end, relevant design decisions need to be guided or constrained in an appropriate way, as described below.

Scenario 2: architecture is the CA

In this second scenario, let us assume that the architecture in Figure 4-1 is the CA, in which case only the white boxes are fully specified, while the shaded and striped ones correspond to variation points of either optional or alternative type. As in the previous scenario, there are two possible cases to consider.

If all the sensitivity points in this architecture are located in mandatory components (which should be the goal of every product line design), then the CA design decisions will address the common quality of all products.

However, the above case is not always what is obtained in reality. Oftentimes, there are two or more sensitivity point localized in both areas that has been fully are not fully specified (variation point). The architects specified (e.g., mandatory components) and areas that can only design to fulfill the quality goal of the mandatory component and expect product architects to fulfill their part in designing the variants for the appropriate quality response. If the teams are different, this may be hard to do without duplication of efforts.

To ensure conformance of the PA design decisions to those of the CA, in order to fulfill a common quality goal, an evolvability point and evolvability constraint pair are needed. It is not every variation point in the CA that is an evolvability point, but only those that interact with the sensitivity point. The designers of the CA will accompany such evolvability points with constraints/guidelines to help product architects in their work.

Evolvability constraint is a statement about an evolvability point that guides product architecture creation in order to fulfill desired quality goals. Just like every other form of constraints, it may be described using the syntax and semantics of an ADL or other constraint language. The constraint may restrict variant components in their interaction protocol, internal states, architectural styles, implementation or usage [9] in order to fulfill some quality goals.

The combined use of evolvability point and evolvability constraints ensures that PAs remains in conformance with the CA. The following is a description of an evolvability point (EP) and its corresponding evolvability constraint (EC), as defined in a recent case study of a product line called btLine, in the domain of mobile and electronic payment systems.

EP: The response time of the btLine product to tasks delegated to it is dependent on whether it is interfaced directly to the legacy and back office systems of its host organisation or not. The fact that design decision on product integration varies from product to product makes it an evolvability point.

EC: To enhance response time for transaction involving a product, external data request from within the product (e.g., balance of a customer account in the host banking system) must not involve complicated and time-consuming queries. Alternatively, an external integration mechanism may be deployed to synchronize account details between the bank systems and their local btLine product; of course with guidance from the btLine team. Better still, outbound request from a btLine product to external systems may be routed to a low-traffic data source or business component for improved response time.

4.4 Conclusion and Open Issues

We highlight the problem context and the challenges of ensuring quality attributes conformance between a product line common CA and its product PAs. Subsequently, we described a technique for implementing this form of conformance during product development and maintenance. The technique focuses on identifying variation points that interact with sensitivity points. Those points, referred to as evolvability points, are accompanied with suitable guidelines and/or constraints. The constraints inhibit any PA design decisions from degrading the preset quality attributes' responses of the CA. Adhering to the constraints and guidelines would ensure that the quality attributes of the PA are in conformance with those of the CA.

The main contributions of this approach include its architecture-centric focus for reasoning about quality attributes conformance of the product architectures to the CA and systematic use of variation points to constrain product architectures from deviating from the preset qualities of the CA. Both of these should facilitate understanding of the interactions, conflicts, and tradeoffs between quality attributes of different forms of architecture encountered in product line development.

Much of the issues relating to quality attributes conformance between the CA and the PAs are still open. First and foremost, considerable advances have been made regarding architecture recovery from existing systems—but extraction of CA and PAs from such systems is still an open area for research.

Second, there is need for characterizing those areas of the CA that do not feature any variation points, but that have the potential of determining qualities both in the CA and the PAs.

Other open questions include: What approach can be used to resolve quality attributes conflicts between the CA and PA? How responsive is the current result to product line development in the evolutionary approach involving reverse engineering or reengineering? What is the impact of the CA evolving in terms of functionality and quality on the quality responses of the product architectures? How can software product line specialists utilize the result of the characterizations of conformance checks between a product line's CA and PAs for checking conformance of the code-dependent (as-built) architecture to the documented (as-designed) PAs? Finally, while tool support is always a plus, the exact details of support for quality conformance checking and traceability in a product line context have yet to be worked out.

Some of these issues will be addressed in our future research.

4.5 References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, 2nd edition, 2002.

[2] J. Bosch. *Design & Use of Software Architectures*. Addison-Wesley, Harlow, England, 2000.

[3] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. *ICSR '98*, p. 308,Washington, DC, USA, 1998.

[4] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures—Methods and Case Studies*. Addison-Wesley, Reading, MA, 2002.

[5] P. Clements and L. Northrop. *Software Product Lines*. Addison-Wesley, Reading, MA, 2002.

[6] P. Clements and L. Northrop. *A Framework for Software Product Line Practice Version 4.2*. Software Engineering Institute, 2005.

[7] E. Dincel, N. Medvidovic`, and A. van der Hoek. Measuring product line architectures. In *PFE '01: Revised* Papers from the 4th International Workshop on Software Product-Family Engineering, pages 346–352, London, UK, 2002.

[8] R. Kazman, M. Klein, and P. Clements. *ATAM: Method for architecture evaluation*. CMU SEI Technical Note CMU/SEI-2000-TR-004, ADA382629, Software Engineering Institute, Pittsburgh, PA, 2000. http://www.sei.cmu.edu/publications /documents/00.reports/00tr004.html

[9] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. *ESEC'97/FSE-5*, pp. 60–76, Zurich, Switzerland, 1997.

[10] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software. Eng. Notes*, 17(4):40–52, 1992.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[12] C. Stoermer, L. O'Brien, and C. Verhoef. Practice patterns for architecture reconstruction. *WCRE '02*, p. 151, Washington, DC, USA, 2002.

[13] C. Stoermer, L. O'Brien, and C. Verhoef. Moving towards quality attribute driven software architecture reconstruction. *WCRE '03*, p. 46, Washington, DC, USA, 2003.

[14] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering. *J. Syst. Software.*, 66(3):225–239, 2003.

5 Identification Of Variation Points Using Dynamic Analysis

Bas Cornelissen
Delft University of Technology
The Netherlands
s.g.m.cornelissen@ewi.tudelft.nl

Bas Graaf
Delft University of Technology
The Netherlands
b.s.graaf@ewi.tudelft.nl

Leon Moonen
Delft University of Technology and CWI
The Netherlands
Leon.Moonen@computer.org

Abstract

In this position paper we investigate the use of dynamic analysis to determine commonalities and variation points as a first step to the migration of similar but separate versions of a software system into an integrated product line. The approach detects forks and merges in different execution traces as an indication of variation points. It is illustrated by a simple implementation, which is applied to an academic example. Finally we formulate a number of research issues that need to be investigated further.

5.1 Introduction

Already many successes have been reported with respect to the use of product line approaches in software development organizations [1]. A company that migrates to a product line approach must define a product line architecture that incorporates the design decisions common to all product line members. Additionally, the variability between the different product line members is to be made as explicit as possible.

In practice, the idea of following a product line approach can be applied in various levels of detail. For example, one can define a reference architecture which specifies all commonalities between products but does not make the variation points explicit. As such, we can distinguish between various maturity levels in a product line deployment [2]. This is also illustrated in an industrial example discussed by Graaf et al. [3].

A typical situation in which the adoption of more product line concepts, and thereby raising the maturity level, is beneficial, is when a company has developed several versions of a product for different customers. All these versions are extended in one or more ways with respect to some original system that was initially developed. At some point a customer comes along that requires some of the extensions that were already implemented, but for different versions of the product, and thus their implementations reside in different development branches. As more versions are being developed, such a situation becomes more and more likely. At that point these extensions should be reengineered into clearly defined, configurable features by making *variation points* explicit, ideally enabling late binding. Domain and application engineering methods have been proposed to solve this problem. Typically these approaches are applied in a context where a product line is developed from scratch, and do not take existing source code into account. However, new product lines are typically not developed from scratch, but evolve from a set of similar, traditionally developed products. Furthermore, often many design decisions are only explicit in the source code. In this paper we consider the problem of detecting forks and merges in the execution traces generated by different versions of a system so as to identify its variation points. The remainder of this paper is organized as follows. Section 5.2 discusses some related work. In Section 5.3 the basic idea of how execution traces can help in identifying variation points is presented. Section 5.4 explains a simple implementation of this idea that detects forks and merges in execution traces. This implementation is applied to a simple example in Section 5.5. The paper is concluded with some discussions and directions for future work in Section 5.6.

5.2 Related Work

Van Gurp et al. [4] provide an excellent introduction to the concepts of variability in software product lines and discuss how variability can be documented using feature graphs. However, they do not discuss in much detail how commonalities and variation points can be discovered.

Approaches for domain engineering aim at identifying commonalities and variabilities for the definition of product line architectures. Scope, variability, and commonality (SCV) analysis discussed by Coplien et al. [5] provides a systematic way of thinking about commonality and variability. The same work also introduces FAST, an approach for domain engineering based on SCV-thinking. Other domain engineering approaches are FODA [6] and FORM [7]. Typically these approaches are based on the analysis of high-level information, such as requirements to identify variabilities and commonalities.

Execution traces have been used for many purposes in the program analysis community. However, in only a few cases traces from different programs were compared to each other. Much of the work is concerned with identifying which components are required for a specific feature or set of features.

The software reconnaissance technique proposed by Wilde and Scully [8] compares execution traces of different sets of scenarios to identify which components are required for a specific feature.

Eisenbarth et al. [9] apply formal concept analysis to execution traces that each exhibit a different feature, so as to identify feature-component relations. As such they also investigate the commonalities and variabilities between different features in terms of the components required to implement them.

These approaches compare different execution traces of the same program. Therefore, they rely on the assumption that the exhibition of a certain feature can be controlled by the user, which is not always the case.

5.3 Tracing and Variation Points

Suppose we have two branches of a software system, one being the base system and the other a variant with one or more additional features. Detection of variation points using execution traces is based on the idea illustrated by Figure 5-1. In this graph, we have projected one trace on top of the other. Each node in the graph denotes the usage of a component for the execution of a scenario. The arcs indicate the order in which the components were used. The fork in Figure 5-1 can be considered the variation point. All behavior executed up to the split is common behavior and the components that are used after the split are feature-specific.

The components considered in an execution trace are units of source code. Different levels of granularity are possible: statement, method, class, package or other abstractions.

Execution traces are obtained by executing some scenario. Comparison of execution traces is only meaningful when the corresponding scenarios are either the same or very similar. In this context a scenario is defined by the input offered to the system. We do not consider the system's response as part of the scenario, as the intention is to execute scenarios on different systems that yield different responses.

For the localization of variation points in the implementation that correspond with the specific features, we need two execution traces: one in which the extension is exhibited and one in which it is not. Depending on the feature, it may or may not be possible for the two scenarios that generate these traces to be identical. In case the exhibition of a certain feature depends on the input, different scenarios are needed. This can be the case, for example, when the feature is only activated when a user clicks a certain GUI button. If activation does not depend on the scenario, we compare execution traces generated by various versions of a system.

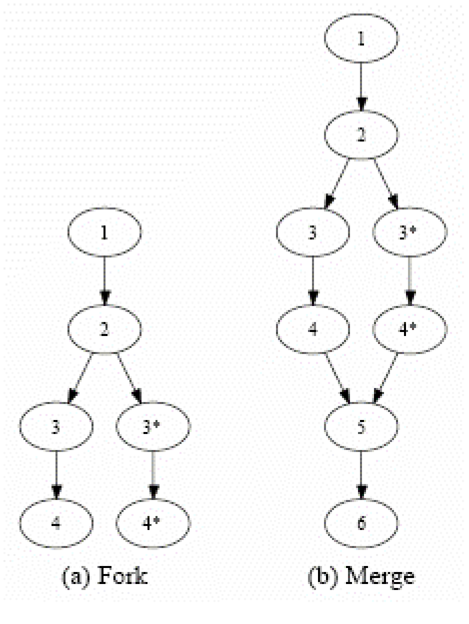


Figure 5-1: Forks and merges in an execution trace

The underlying assumption in our approach is that both execution traces will largely resemble each other and the associated graphs will have most nodes in common, up to the point where the additional feature is exhibited (Figure 5-1). Automatic detection of such a fork is trivial: we take the node before the first deviation in the two execution traces. The detection of a merge, however, is more involved. Simply detecting the first pair of nodes that are identical after the fork might not be meaningful. Usage of a specific component in both traces does not necessarily imply that the same behavior was demonstrated from a user's perspective. The next section will describe a solution to this issue using an evolving comparison window.

The generation of traces that can be compared meaningfully is even more complicated if non-deterministic behavior is considered (e.g., in games).

5.4 Approach

In this section, we first present the running example that is used in the remainder of this paper to illustrate our approach for the identification of variation points using execution traces. Next, we explain how we obtain those traces and finally how they are processed.

5.4.1 Running example: Pacman

The system we use as a running example in this paper is a java-based game called Pacman. With a little imagination, we can regard Pacman as a simple example of a software product line.

Pacman is a modest software system consisting of 20 java classes and approximately 1000 lines of code. Like in a software product line there exist several variants of this system, each with distinct added features.

For example, in the reference system there is one hardcoded map being loaded whenever a game is played. In another version, which can be considered a member in our product line, functionality has been added (in a separate class) to read user-defined maps from a file. Yet another version of the system features an additional type of entities on the map with which the player and the monsters can interact.

5.4.2 Dynamic analysis using aspects

We obtain execution traces by instrumenting the system with trace statements. We add these trace statements by means of aspect-oriented programming. Aspect-oriented programming is extremely suitable for implementing a crosscutting concern such as tracing since it allows us to add code at various program locations with limited effort. We use AspectJ to weave additional code in the system such that, whenever a method is called, a message is printed to a log file. This message contains both the method being called and the class to which this method belongs.

Now, we can generate traces containing the methods called during execution. Depending on the desired level of granularity of variation point detection, we may need to further process this trace, e.g. to generate a trace on the class level.

Alternatively, one could use the Java Virtual Machine Profiler Interface (JVMPI) to collect traces from a system, as is done, for example, by Reiss and Renieris [10].

5.4.3 Determining variation points

When dealing with software product lines, each of the product line members generally contains a set of features. Typically, the members have some of these features in common whereas others are product-specific. If an architect is to combine two or more products, the components responsible for the latter type of features—the *variation points*—must be determined.

We propose a method in which we compare the traces generated by two versions of a similar system to discover variation points. On the one hand we have a trace generated by the reference system, called the *reference trace*, and on the other hand a trace generated by an extended version, called the *feature trace*. These traces are to be obtained by running both systems using similar scenarios: ideally, the latter differs from the former only in that the specific extension is exhibited.

As mentioned in Section 5.3, branches are not necessarily considered merged as soon as the two traces once again have one method in common. For this reason, we require the traces to have *multiple* consecutive methods in common.

The algorithm being applied reads as follows:

1. Compare the traces of the reference system and the product line member line by line.
2. If the two methods at hand differ, the traces have **split** into branches. Create an *N*-size checksum of the current reference method and the next *N-1* methods (henceforth, we will call this the *reference window*).
3. Next, create a checksum of the upcoming N methods in the feature trace, thus creating the *feature window*.
4. If the checksums are equal, the branches are considered to have **merged**. If they do not match, shift the feature window down one method, thus creating a new feature checksum. Repeat this step a maximum of *M* times.
5. If the checksums still do not match, shift the reference window down one method. Repeat the previous step, but repeat the current step a maximum of *M* times.
6. If there is still no match, either *M* is too small or the branches never merge, i.e. the systems never again exhibit the same behavior at the method level.

The values for *N* and *M* are variable and depend on several factors. In assigning suitable values to these variables, important factors include the size of the system and the predicted impact (in terms of the amount of associated method calls) of the feature at hand. We expect the architect to have sufficient knowledge of the system at hand to choose appropriate values for *M* and *N*.

The branching behavior derived by the algorithm can be visualized by presenting contexts (of predefined sizes) of all forking and merging points in the traces to the user. By visualizing and inspecting the branching behavior, the architect has a way of identifying which methods and classes account for member-specific features. Having approximated these variation points, it takes much less effort to merge the two versions than if the entire systems had required close inspection.

5.5 Preliminary Results

To illustrate the method presented in the previous section we have conducted some experiments on the Pacman system described earlier.

In this section, we will highlight the experiment involving Pacman's reference system and the modified version featuring separate map handling.

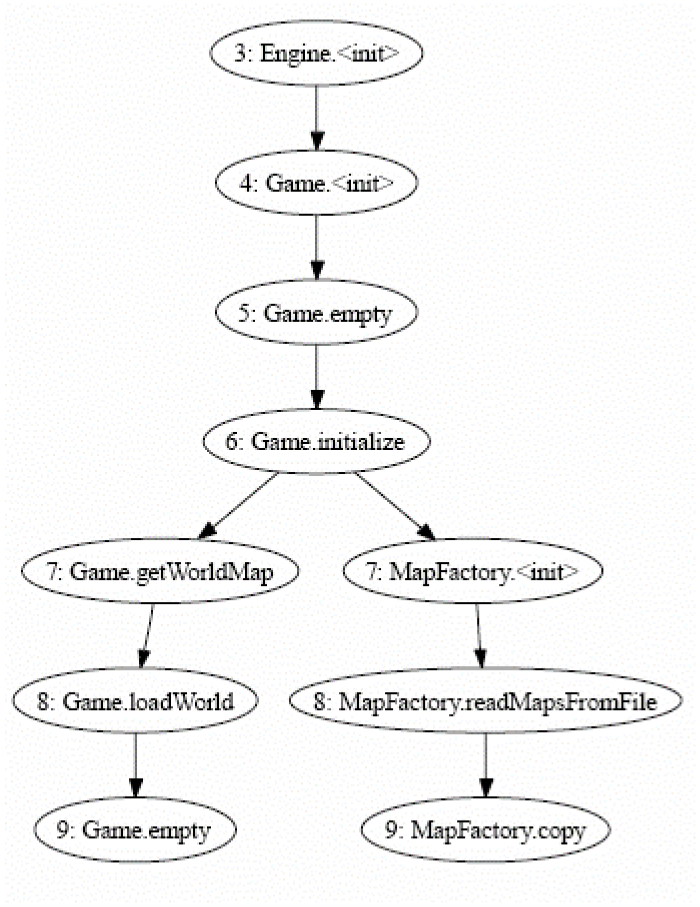


Figure 5-2: A fork and its context in the trace.

5.5.1 Generating traces

Choosing appropriate scenarios is relatively easy in this case, as loading maps is part of the initialization phase and therefore not subject to human intervention. It is simply a matter of running both programs and exiting without actually having played game.

Part of a method trace as generated by use of the aspect mentioned in Section 5.2 is depicted in Listing 5-1.

Incorporation of the actual stack depths is not part of the results discussed here and is subject to future research.

```
Pacman.main
Pacman.<init>
Engine.<init>
Game.<init>
Game.empty
Game.empty
Game.<init>
Game.initialize
...
```

Listing 5-1: Part of a method trace

5.5.2 Branching behavior

We are now ready to compare the traces by using the algorithm described in Section 5.4.3. However, we need to define some parameters first.

Since we are considering a small system and a not so complicated feature, we do not expect branches to be very long, e.g. perhaps tens of methods at most. For the same reason we will set the checksum size at a relatively small value, e.g., 5 methods. Finally, the size of the context being presented to the user is set to 7.

The results can be viewed in Figure 5-2 and Figure 5-3. Figure 5-2 depicts the context of the point where the feature trace started deviating from the reference trace. One can easily see that whereas in the original version a local method is invoked to get a map, the other version instantiates a whole new class that deals with the map handling.

Figure 5-3 illustrates that not many methods calls later, the branches have merged. From here on, the traces are apparently similar.

Judging by the visualizations—if presented at the correct part and, if desired, migrate the components associated abstraction level—an architect can easily isolate the feature specific with this variation point towards other existing product line members.

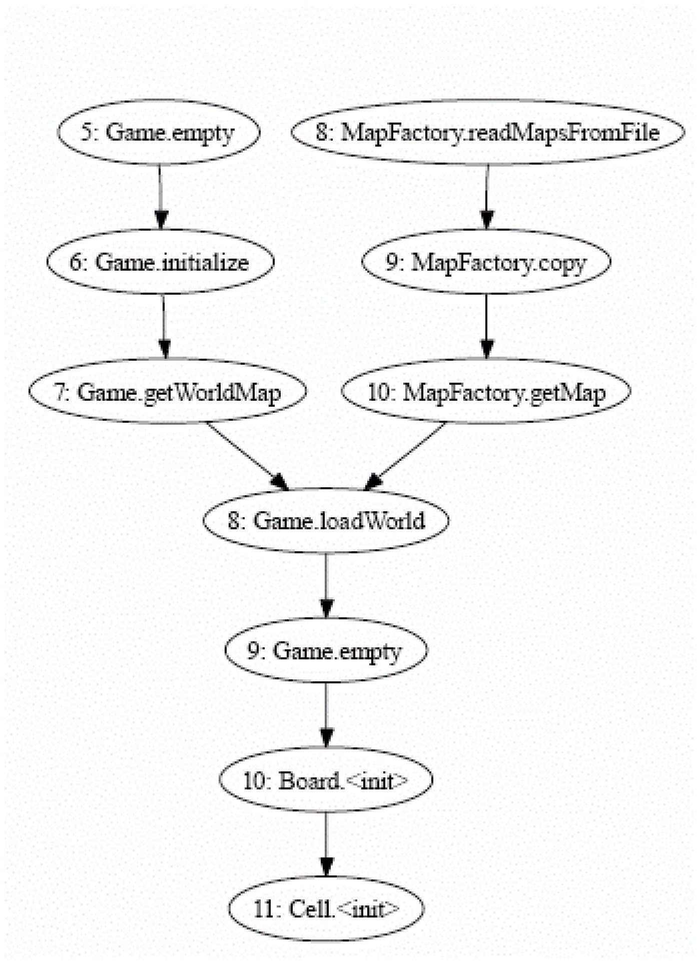


Figure 5-3: A merge and its context in the trace.

5.6 Discussion and Future Work

Effort. To repeat our experiment on a different subject system, one can apply the following process:

1. Perform a quick (1-hour) exploration of the system to gain some insight in its structure. This provides an initial estimate for the values of the *M* and *N* parameters.
2. Determine appropriate scenario(s) that exercise the desired features.
3. Add tracing instrumentation to the system, e.g. by weaving aspects.
4. Collect execution traces for given scenarios and (automatically) compare them to find variation points.
5. If desired, repeat step 4 using alternative values for *M* and *N* to fine-tune the results.

Precision. In the current implementation we more or less assume that a merge point is not located arbitrarily far from a fork. Hence, we introduced the *M*-parameter in our detection algorithm. This assumption is valid because we require that one version is a strict extension over the other.

If we abandon this requirement, we would have to search *both* execution traces all the way to the end to find potential merges. The complexity of this search is $O(n^2)$, which could be problematic for systems of realistic size, involving traces consisting of millions of components. This is why we advocate a sensible value for M: a value defined by the architect, based on how much impact he expects the particular feature to have on the given trace granularity level (method level in the case presented here). In the future we may be able to automatically determine optimal values for specific systems.

Future work. To render identification of variation points feasible in the case of complex systems, we need more refined techniques. One approach is to take into account not only the methods being called but also their actual parameters. This would require a straightforward extension of the tracing instrumentation.

Another option is to also look at the stack depth or maybe even the complete stack whenever a method is called. Both these extensions to our technique potentially allow for the detection of extra forks, and increase the probability that an identical entry in the two call traces indeed implies that the two versions were again exhibiting common behavior, from a user's perspective. Probably this also means that the N-parameter can be smaller, which in turn reduces the cost of the checksum calculations.

An alternative approach in dealing with systems of realistic size would be to not directly analyze the method trace, but to first lift its elements to higher levels of abstraction, e.g., from methods to classes or packages. To this end, we would first have to extract information with respect to the structural decomposition of the system.

Finally, once a feature is localized a next step is to modularize the code that implements it. To provide guidelines for this step we will investigate whether the number of times two traces intersect (in terms of identical methods being called) before the same behavior is exhibited (as defined by the N-parameter) could be a measure for the degree of 'crosscuttingness' of a feature, and hence for the effort required to reengineer such a feature into a reusable component.

5.7 Acknowledgements

This work was partially supported by NWO Jacquard project Reconstructor.

5.8 References

[1] Software Engineering Institute. Product Line Hall of Fame. http://www.sei.cmu.edu/productlines/plp_hof.html, 2005.

[2] Jan Bosch. Maturity and evolution in software product lines: Approaches, artifacts and organization. In *Proceedings of* the Second International Conference on Software Product *Lines (SPLC 2)*. Springer-Verlag, August 2002.

[3] Bas Graaf, Hylke van Dijk, and Arie van Deursen. Evaluating an embedded software reference architecture – industrial experience report. In *Proceedings of the 9th European* Conference on Software Maintenance and Reengineering *(CSMR)*, Manchester, UK, March 21-23 2005. IEEE Computer Society.

[4] Jiles van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in

software product lines. In *Proceedings* of the Working IEEE/IFIP Conference on Software *Architecture(WICSA'01)*. IEEE Computer Society, August 2001.

[5] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November 1998.

[6] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990. http://www.sei.cmu.edu/publications /documents/90.reports/90.tr.021.html

[7] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, January 1998.

[8] N. Wilde and M.C. Scully. Software reconnaissance: Mapping program features to code. *Journal on Software Maintenance: Research and Practice*, 7(1):49–62, January 1995.

[9] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society, May 2001.

[10] Steven P. Reiss and Manos Renieris. Generating Java trace data. In *Proceedings of the ACM 2000 conference on Java Grande*. ACM Press, 2000. ISBN 1-58113-288-3.

6 Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product line Migration[1]

Dharmalingam Ganesan, Jens Knodel
Fraunhofer Institute for Experimental Software Engineering (IESE),
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{ganesan, knodel}@iese.fraunhofer.de

Abstract

Domain-specific reuse is seen as a promising way to increase the value of reuse. This paper reports our ongoing work aimed at identifying domain-specific software components from an existing system to achieve large-scale reuse. The fundamental motivation of the proposed method is to reduce the amount of source code the human-expert has to explore in order to identify domain-specific candidates. The basic premise assumed by this method is that reusable components have certain quality attributes like functional usefulness, readability, testability, etc., and which can be measured to certain extent with help of metrics.

Keywords: domain engineering, reuse, reverse engineering, metrics, software product lines

6.1 Introduction

Software reuse is considered as a promising way of developing systems. It helps an organization to improve their productivity and the quality. Software reuse can be applied to any life cycle product, not only to source code. Jones [10] identifies ten potentially reusable aspects of software projects as shown in Table 6-1. (Ordering of aspects in Table 6-2 is not with respect to priority.)

1. architecture	6. estimates
2. source code	7. human interfaces
3. data	8. plans
4. design	9. requirements
5. documentation	10. test cases

Table 6-1: Reusable Aspects of Software Projects

However, in practice, granularity of reuse is small. That is, very often, utility libraries (for e.g., string, math libraries) are reused across products. Value of such a re-

1 This work is partially funded by German ministry under EUREKA 2023/ITEA-ip00009 'FAct based Maturity through Institutionalization Lessons-learned an Involved Exploitation of System- family engineering' (FAMILIES).

use is quite limited [8]. In a mature domain, most of the required solutions already exist in current implementations. It has been argued in [13] that there are three categories of software that make up a system:

- **Utility components** contribute to 20% of whole application size.
- **Domain-specific components** contribute to 65% of whole application size.
- **Application-specific components** contribute to 15% of whole application size.

The distribution shows that reuse of domain-specific components from an existing system has the most potential in reducing the development cost and maintenance effort [1]. The identification of domain-specific components is not an obvious task since systems are typically developed for a single customer. Designers and engineers thereby do not distinguish between domain-specific and application-specific components [9] as it is explicitly done in product line engineering. So these components are not organized separately. Hence, expert effort has to be spent to make these components apparent.

We believe that reverse engineering can help to identify domain-specific components and therefore to support the reuse activities by reducing the expert effort in searching for component candidates, which is a problem especially for large systems. Our approach helps experts to semi-automatically identify domain-specific components. From here onwards, we limit our discussions only to object-oriented (OO) systems. And consequently, the term component refers to the collection of functionally related classes with specification of required and provided interfaces.

The fundamental motivation of the proposed approach is to reduce the amount of data the human expert has to review in order to identify domain-specific classes. The basic premise assumed by this method is that reusable classes have certain quality attributes like functional usefulness, readability, testability, etc. These quality attributes are mapped on metrics (e.g., by using the GQM method). The method classifies the domain-specific classes based on the metrics derived. The expert has to validate only a few number of proposed candidates, which, if accepted, become then the foundation of reusable components (see Figure 6-1).

The remainder of the paper is organized as follows: Section 6-2 explains the factors affecting reusability. Section 6-3 presents component extraction method. Section 6-4 presents the related work, while Section 6-5 concludes this work.

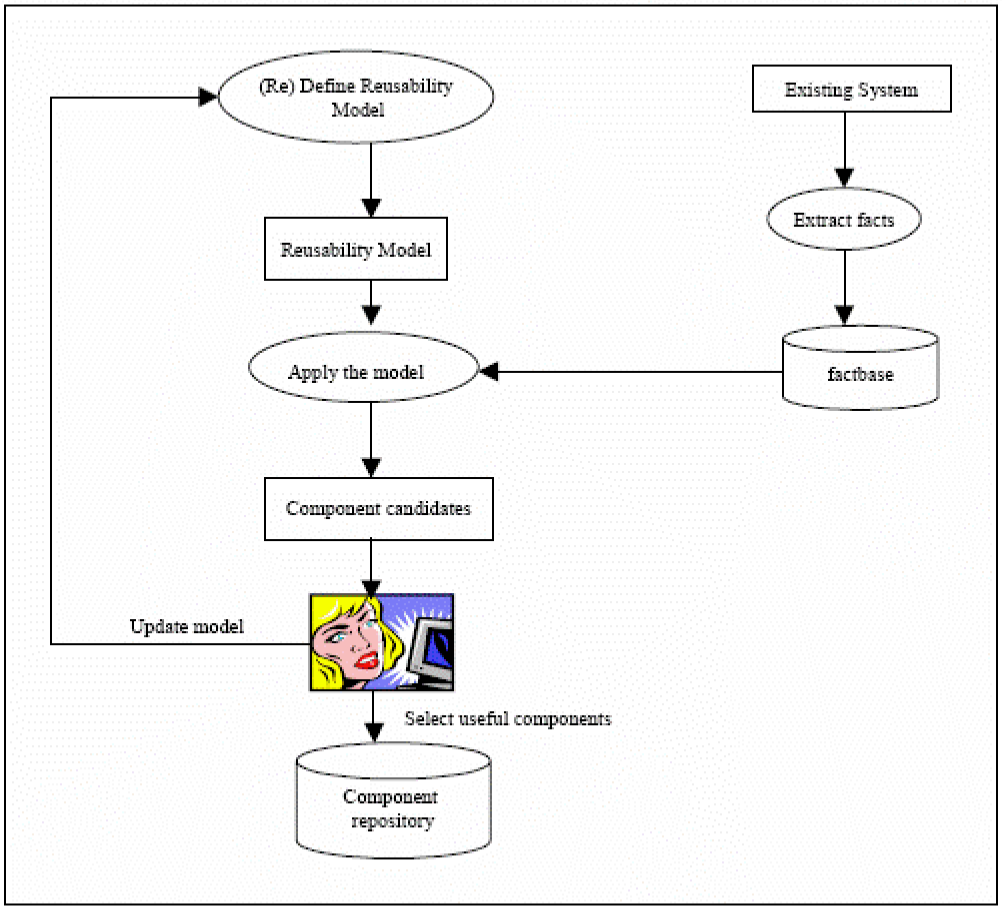


Figure 6-1: Process model for component extraction

6.2 Approach

6.2.1 Terminology

Forward Engineer: An engineer who wants to reuse existing components of the same or similar domains to reduce development effort.

Domain Expert: A specialist with detailed knowledge of the domain who is also familiar with the architecture of the system where the existing components reside.

Reverse Engineer: A person having understanding of OO metrics and being capable to analyze an existing system (no need to have expertise in the domain).

6.2.2 Factors Affecting Reusability

Figure 6-2 shows a “fishbone diagram” that represents the factors affecting reusability. It can be observed from this figure that reusability depends on *Usefulness*, *Costs* and *Quality*. Each of these factors is explained below.

Usefulness

To be reused, a prerequisite is that the component implements functionality that is useful for the new system. It is extremely hard to decide in an automated way whether or not a component will be useful in a new system, since this decision is based on domain knowledge and the requirements of the new system. However, an indirect automatable measure of usefulness was developed to measure the reusability of the existing component within the analyzed system itself (i.e., its origin). The assumption is that the highly used components within a system are a good candidate for reuse in a new context.

There is also a limitation because of our assumption: We tend to exclude those domain-specific components that are not frequently used in the existing system. It is important to note that the domain expert is crucial to decide about the usefulness of a component candidate.

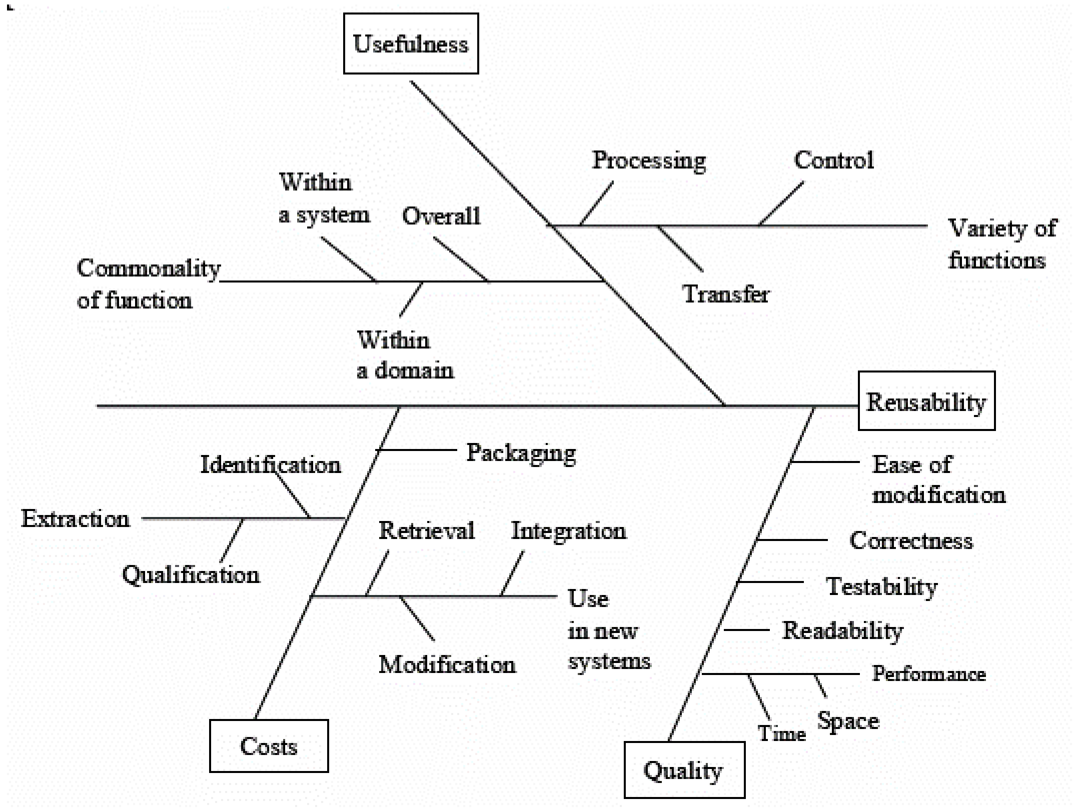


Figure 6-2: Factors affecting reusability [4]

Cost

Reuse cost includes cost of identifying a component from the existing system, modifying and integrating them into a new system. Measures of size and complexity of a component provide a partial indication of difficulty in adapting it to reuse in a new system. The cost to reuse the component is influenced by the readability of its code, a characteristics that can again be partially evaluated using size and complexity measures. That is, small and simple code fragments are usually easier to read and adapt than larger and complex fragments.

Quality

The quality of the component is important in order to succeed in reuse-driven development. Several qualities that are important for component reuse are correctness, readability, testability, ease of modification, and performance, but most of them are not directly measurable. Measures of size and complexity of a component however provide a partial indication of the presence of these qualities in it.

6.2.3 Metrics for Measuring Costs, Usefulness and Quality

Table 6-2 contains definitions of the metrics used for measuring costs, usefulness, and quality. A complete discussion about these metrics can be found in [5]. Our motivation is to come up with a reusability model, which contains metrics and the suitable upper and lower bounds to support identify reusable classes.

Metric	Definition
NMPUB	The number of public methods implemented by a class.
WMC	Cyclomatic complexity of a class.
NOC	The number of children/grandchildren of a class.
DIT	The level a class is located from the root in the inheritance hierarchy.
OCAEC	The number of times a class has been used as an attribute in other classes.
CALLS_IN	The total number of times the methods of a class was called by other classes.
LCOM	Cohesion—The number of sets of methods that access the same attributes.

Table 6-2: Definition of Metrics

Measuring Usefulness

We measured the functional usefulness using the assumption: a class that is used frequently within a system is a good candidate for reuse in a new system in similar domain. In OO systems, a class A can use another class B in the following ways:

- Methods of A invokes the methods of B
- A contains an instance of B as its attribute
- A inherits from B
- A can read/write attributes of B

Hence, we have chosen the metrics namely NOC, OCAEC, CALL_IN and DIT to measure the usefulness within the existing system itself.

If a class has many children/grandchildren, then it is likely that it implements certain generic functionality. Hence we need to take only the lower bound for NOC because the more the number of children/grandchildren, the higher is its assumed genericity.

The reason for choosing DIT metric is that if a class occurs near the leaf of the inheritance tree then, in most cases, it implements probably certain specialized functionality. For reuse candidate's identification, specialized functionality is obviously not the first priority. That is, we should not go too deep in the inheritance hierarchy. Hence we need to take only the upper bound for DIT metric.

In many applications, classes are not always in the inheritance tree. That is, there are classes that don't have either a parent or children and such classes might also be good candidates for reuse. In order to include such classes for potential reuse, we have chosen CALL_IN and OCAEC. CALL_IN metric is used to identify those classes that are used heavily by methods of other classes. The more the value of CALL_IN, the higher is its usefulness within the system. If the value of CALL_IN is below a certain value, it is likely that its services are not important that to the system. Hence we need to take only the lower bound for CALL_IN. OCAEC can be used to measure how useful the class is in building the other classes. That is, if a class has higher OCAEC then it is used an attribute in many other classes. Similar to CALL_IN, we need to select only the lower bounds for OCAEC.

Measuring Cost

We can measure the reuse costs using the metrics NMPUB and WMC. If NMPUB and WMC are high then it might take more effort to understand, modify and integrate them into a new system. On the other hand, if both NMPUB and WMC are too low, it is very likely that there is nothing interesting in it. So it is better not to exceed both the bounds.

Measuring Quality

We can measure quality using NMPUB, WMC and LCOM. If a class has high NMPUB then it is likely to have impact on correctness, readability. Testability can be partially predicted with help of WMC [3]. Higher the WMC, lower is the testability of a class. If cohesion metric LCOM is high, it is very likely to reduce the understandability and readability of the class because of the variety of functionality implemented in it. Hence, only the upper bound is necessary for LCOM.

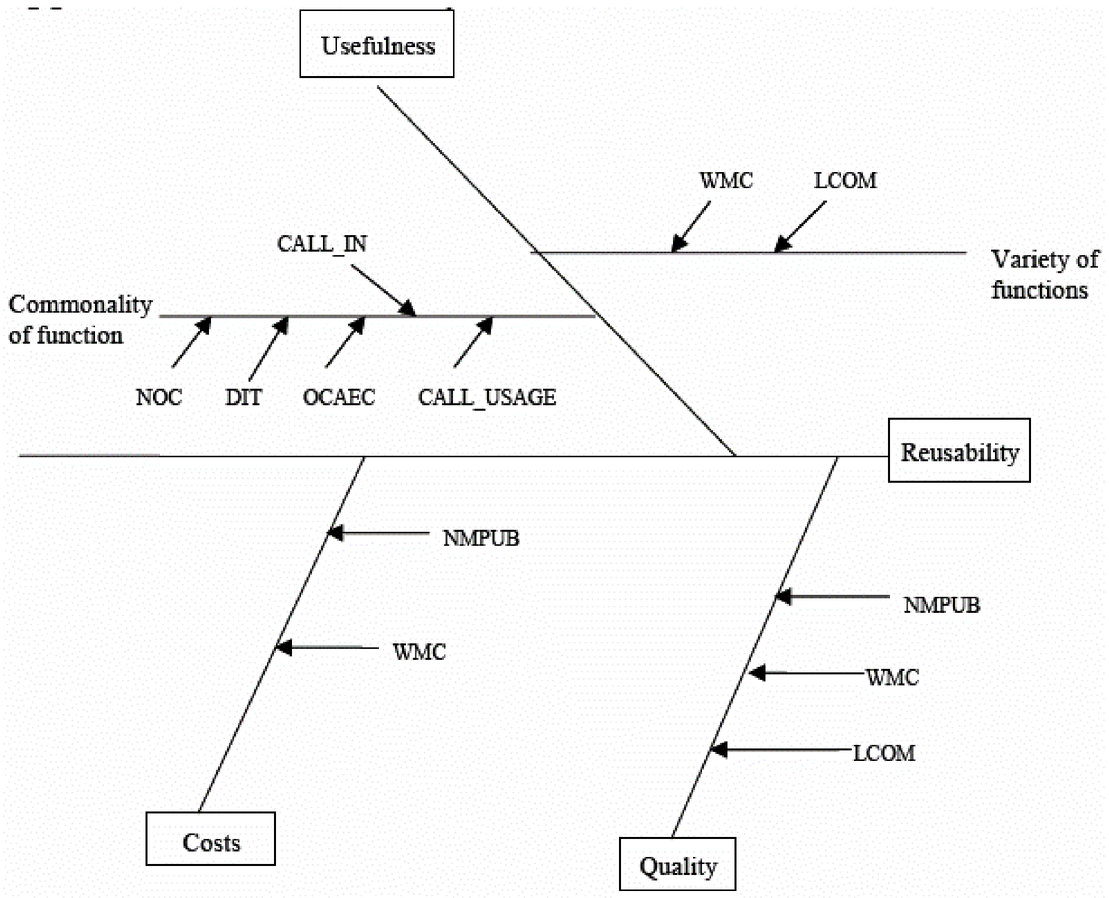


Figure 6-3: Associating OO metrics with factors affecting reusability

6.3 Process

We introduce an approach for the extraction of domain-specific components from an existing system, where reverse engineers, forward engineers, and the domain experts work closely together. Figure 6-4 depicts the 10 steps of our approach in the pseudo-code format.

Step 1: Goal Description: In this step, the forward engineer formulates the goal and explains it to the reverse engineer. The forward engineer can describe the kind of components he wants to reuse from the existing system. For instance, let us assume that the forward engineer wants to build an IDE for modeling software architectures by reusing an existing IDE for Java. The forward engineer then explains the need for components implementing functionality related to workspace, projects, package hierarchy, and file management.

Step 1: The forward engineer *describes the goal* to the reverse engineer.

Step 2: The reverse engineer *sets up the fact base*.

Step 3: The reverse engineer *selects metrics* and choose its bounds.

Step 4: The reverse engineer *identifies candidate classes* which satisfied the criteria defined in step 3.

Step 5: A *"lightweight" review* on the classes from step 4 is done by reverse engineer. If he is *not satisfied* then he goes back to step 3. Otherwise, he passes the candidates to step 6.

Step 6: The domain-expert *reviews the candidates and classifies* them.

Step 7: The reverse engineer *analyses the classification* made by the domain-expert.

Step 8: Both the engineers start *building components* from the *key classes* of step 6.

Step 9: *Interface analysis* is done by the reverse engineer to know the dependency between the components from step 8 and the rest of the existing system.

Step 10: The forward engineer makes the *final decision* about the reuse of the components using the output of step 9.

Figure 6-4: Different steps for component extraction

Step 2: Setting up the fact base: The reverse engineer parses the source code of an existing system and builds an initial model of source code. The initial model could be, for example, an RSF representation of the source code. In addition, for each class in the source model, he computes the metrics defined in Table 6-2.

Step 3: Select metrics and choose its bounds: In this step, the reverse engineer chooses bounds for the metrics defined in Table 6-2. But the problem is a lack of an analytical method that a reverse engineer can use to choose the bounds for these metrics. In the first iteration, in order to choose bound(s) for a metric, he computes the average of the metric values. This seems to be like a trial and error but it is nevertheless a meaningful starting point.

Metric	Minimum	Maximum
NMPUB	X	X
WMC	X	X
LCOM		X
CALLS_IN	X	
DIT		X
NOC	X	
OCAEC	X	

Table 6-3: Metrics with Lower and Upper Bound

Step 4: Identify candidates: In this step, the reverse engineer applies the metric criteria developed in step 3 to all the classes in the fact base. Classes which satisfied the criteria will be passed to the step 5.

Step 5: Lightweight review: One of the major problems is that reverse engineers usually don't have expertise in the application domain. Therefore he cannot review the candidates identified in step 4 for its usefulness in a new system. But the reverse engineer can do a lightweight review to help the forward engineer. For example, if the number of candidates identified in step 4 is too high then he redefines the criteria defined in step 3. The reverse engineer also uses the goal description during the light-weight review of the identified candidates.

Step 6: Review by the domain expert: In this step, the domain expert reviews the classes identified in step 5 (based on the goal description of step 1). The main focus of the domain-expert in this review is to decide about the functional usefulness of the candidates. Domain expert classify each of the classes given by reverse engineer as follows:

- *Utility* – Classes which implement general utility (for example, math routines).
- *Application-specific* – Those classes that implement functionality specific to single instance of a product line.
- *Domain-specific* – Those classes that contain generic functionality needed for all instances of a product line.

Step 7: Analyze classification: It is important to keep in mind that domain-experts are usually busy. Therefore, the reverse engineer must minimize the amount of the candidates the expert has to review but at the same time maximize the domain-specific candidates. To achieve this goal, the reverse engineer has to analyze the classification of the candidates by the domain-expert. In order to provide the domain-expert with many domain-specific applies a *filtering strategy*. Filtering is used to reduce the search-space for domain-specific classes. That is, certain classes that are most likely not to be domain-specific, are ignored:

- If the root of inheritance tree is not domain-specific then it is likely that the complete inheritance tree is not domain-specific. So, we can filter all the classes involved in such trees. However, this strategy needs to be applied carefully: For example, in Java, the class "Object" is the root class of all classes, but we can develop new applications based on the existing Java classes.
- If a class C is an application-specific/ utility class, then all the classes that are *dominated* by C are likely to be application/utility class. Domination is defined using the dominance tree where the nodes are classes and the edge is the call relation between the classes. Note that this assumption is not always true; there could cases where the application class uses a domain class. Nevertheless, we try to reduce the search-space by making such kind of assumptions.
- If a class C, which satisfied the bounds of the metric OCAEC, is application-specific/utility then all the classes that are used as attributes within the class C are likely to be application specific.

- One obvious filtering strategy is filtering those classes which were already reviewed by the expert. Before he applies the criteria defined in step 3, these reviewed classes can be filtered out.

Step 8: Component building: In this step both forward and reverse engineer works together to build components from the key classes that are identified by step 6. From the key classes, all the required dependencies have to be extracted so that components can be built. This requires analyses of interfaces of the key classes.

Step 9: Interface Analysis: In this step, the reverse engineer analyzes the dependency between the components from step 8 and the rest of the system [11]. By using the factbase, interface analysis identifies all the required interfaces that are necessary for the execution of a component in a new system.

Step 10: Final decision and code analysis: In this step, using the output of the interface analysis, the forward engineer decides whether to reuse the component or not. His decision is influenced as well by factors such as performance.

6.4 Related Work

Basili and Rombach [2] describe a comprehensive framework of models, model-based characterization schemes, and support mechanisms for better understanding, evaluating, planning and supporting all aspects of reuse. We follow their reuse-oriented software environment model to set up a component repository for product line migration.

Caldiera and Basili [4] describe *Care* that helps identifying reusable component using a user-defined "reusability attribute model" based on software metrics. We customized this approach to object-oriented paradigm to support the product line migration activities in the presence of existing systems.

Dunn and Knight [6] describe a model based on an expert-system for the identification of reusable components from existing systems. Suitability of this expert-system to object-oriented paradigm needs further research.

Diaz and Freeman [12] describe a scheme to classify software for reusability. Their premise is that reuse can happen only when there is an automatic way of retrieving the required software components from the repository. Introducing such a classification scheme is a part of our future work.

Etzkorn and Davis [7] describe an approach for automatically identifying reusable classes from object-oriented system. Their PATRicia system uses reusability metrics and a quality model defined by user to identify reusable classes. Their CHRis tool uses natural-language techniques to help expert deciding whether a class implements certain useful functionality.

6.5 Conclusion and Future Work

In this position paper, we described our ongoing work aimed to identify domain-specific reusable components. The fundamental motivation of the proposed method is to reduce the effort spent by the human-expert to identify domain-specific component candidates. The basic premise assumed by this approach is that reusable components have certain quality attributes like functional usefulness, readability, testability, etc. that can be broken down (at least indirectly) into are measurable items.

Our immediate next step is to apply the proposed approach on large-scale systems to identify the benefits and limitations and to base the default boundary values for the metrics on the experiences we will make.

6.6 References

[1] J. Bayer et. al. PuLSE: A Methodology to Develop Software Product Lines, in Proc. of the *Fifth ACM SIGSOFT Symposium on Software Reusability* (SSR'99), ACM, Los Angeles, CA, USA, May 1999.

[2] V.R. Basili and H.D. Rombach. Support for comprehensive reuse. *Software Engineering Journal*, September 1991.

[3] M. Bruntink and A. van Deursen. Predicting Class Testability Using Object-Oriented Metrics. *In Proc. of The Fourth IEEE* International Workshop on Source Code Analysis and *Manipulation; (SCAM 2004)*, pages 136-145, 2004.

[4] G. Caldiera and V.R. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 61-70, February 1991.

[5] S.R .Chidamber and C.F. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Engineering*, 20(6), 476-493, June 1994.

[6] M.F. Dunn and J.C. Knight. Automating the Detection of Reusable Parts in Existing Software. *Proc. of ICSE*, 381-390, 1993.

[7] L.H. Etzkorn and C.G. Davis. Automated object-oriented reusable component identification. *Journal of Knowledge-Based systems*, 9:517-524, 1996.

[8] W. Frakes and C. Terry. Software Reuse: Metrics and Models. ACM Computing Surveys, 28(2), 1996.

[9] B. Graaf, M. Lormans, and H. Toetenel. Embedded Software Engineering: The State of the Practice. *IEEE Software*, 20(6):61-69. November 2003.

[10] C. Jones. Software return on investment preliminary analysis. *Software Productivity Research, Inc.*, 1993.

[11] J. Knodel. On Analyzing the Interfaces of Components. *Workshop in Software Reengineering*, Bad Honnef, Germany, 2004.

[12] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, January 1987.

[13] J.S. Poulin. Measuring Software Reuse: Principles, Practices, and Economic Models. *Addison-Wesley* (ISBN 0-201-63413-9), Reading, MA, 1997.

7 Analyzing the Product Line Adequacy of Existing Components

Jens Knodel
Dirk Muthig
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany
{knodel, muthig}@iese.fraunhofer.de

Abstract

In most cases, adaptation is required to make existing components suitable to the context defined by a product line architecture. This paper presents experience on analyzing the product line adequacy of an existing component in an industrial context. Product line adequacy is based on the results of the application of diverse reverse engineering techniques (architecture evaluation, clone detection, code metrics, and source code analysis). The paper presents these activities, their results, and the action items derived to integrate the component into the product line context.

Keywords: ADORE, product line architecture PuLSE-DSSA, reengineering, reverse engineering.

7.1 Introduction

Product lines are sets of software-intensive systems sharing a set of features and are derived from a common set of reusable assets [6]. Central artifacts of product lines are their architectures, which embrace decisions and principles valid for each derived variant. Hence, architecture development must ensure the achievement of organizational and business goals, functional and quality requirements.

Components as part of product line architectures are explicitly developed for systematic reuse. That is, they must support the scope of variability required and be flexible towards the anticipated changes. Migrating existing components into product line components thus means (in addition to resolving potential architectural mismatches and improving the internal quality) injecting the required variability support. Existing components therefore require a certain amount of adaptations to achieve sufficient product line adequacy. Product line architects face difficult decisions whether to invest in the migration of existing components or to construct new product line components from scratch. Hence, they pass on requests to reverse engineering to analyze the product line adequacy of the existing components. If decided to adapt it, reengineering activities eventually are conducted to prepare the existing component for its use in a product line context.

In this paper, we present a particular case of such a decision by reporting on the analysis of an existing component in an industrial context, where we applied Fraunhofer PuLSE™ (Product Line Software Engineering)[1] [2] and Fraunhofer ADORE™ (Architecture- and Domain-Oriented Reengineering).[2]

The paper is structured as follows: Section 7.2 gives context information on the case study, while Section 7.3 presents the applied approach. Section 7.4 reports on results of the applied techniques; Section 7.5 concludes the paper.

7.2 Context

The case study was conducted in a large organization migrating towards product line engineering following the PuLSE method. The organization defined a product line architecture for a family of multimedia systems in the automotive domain. The products consist of two major parts: a panel (mainly used for user interaction) and the back-end system (mainly used for computation, network functionality, and external media).

The subject component of our case study is one of the key components of the panel. This Graphics component is responsible for the complete interaction between backend and panel, as well as composition and visualization of the exchanged elements. The user interface is based on predefined masks. A mask is thereby defined as a collection of graphical elements and positioning information (e.g., text fields, bitmaps, buttons, lists, labels). The graphical elements contributing to a mask are divided into static information relevant for the panel only and dynamic sequence control information coming from the back-end system. The main architectural driver is the minimization of the data flow between the two parts.

7.3 Approach

The case study combined two Fraunhofer methods: PuLSE, in particular its architectural component PuLSE-DSSA (Domain-Specific Software Architecture), and ADORE.

7.3.1 PuLSE™-DSSA

PuLSE-DSSA deals with product line activities at the architectural level. Since greenfield scenarios [6] are found only rarely in industrial contexts, it is designed to smoothly integrate reverse engineering activities into the process of developing a product line architecture. The main underlying concepts of the PuLSE-DSSA are:

- Scenario-based development in iterations that explicitly addresses the stakeholders' needs.
- Incremental development, which successively prioritizes requirements and realizes them.
- Direct integration of reengineering activities into the development process on demand.
- View-based documentation to support the communication of different stakeholders.

The main process loop of PuLSE-DSSA consists of four major steps (see Figure 7-1).

[1] PuLSE is a registered trademark of Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany.

[2] ADORE is a registered trademark of Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany.

Planning: The planning step defines the contents of the current iteration and delineates the scope of the current iteration. This includes the selection of a limited set of scenarios that are addressed in the current iteration, the identification of the relevant stakeholders and roles, the selection and definition of the views, as well as defining whether or not an architecture assessment is included at the end of the iteration.

Realization: In the realization step, solutions are selected and design decisions taken in order to fulfill the requirements given by the scenarios. When selecting and applying the selected solutions, an implicit assessment regarding the suitability of the solutions for the given requirements and their compatibility with design decisions of earlier iterations is made. A catalog of means and patterns is used in this phase. Means are principles, techniques, or mechanisms that facilitate the achievement of certain qualities in an architecture whereas patterns are concrete solutions for recurring problems in the design of architectures.

Documentation: This step documents architectures by using the organizational-specific set of views as defined in the planning step. It thereby relies on standard views as, for example, defined by Kruchten [8] or Hofmeister [7], and customizes or complements them by additional views.

Assessment: The goal of the assessment step is to analyze and evaluate the resulting architecture with respect to functional and quality requirements and the achievement of business goals. In an intermediate state of the architecture, this step might be skipped and the next iteration is started.

PuLSE-DSSA results in product line architectures documented in a selection of architectural views.

7.3.2 ADORE™

The architecture development yields product line components that have to be engineered. Different components can be engineered concurrently since the product line architecture has defined the component communication, specified the required interfaces, and distributed the responsibilities among the components. In a migration context from single system development, this allows the identification of existing components in the domain that already fulfill the functional requirements completely or at least partially achieve them.

To decide about reusing such existing components, the component's internal quality and suitability for the product line have to be evaluated. It has to be ensured that the component is able to serve the product line needs.

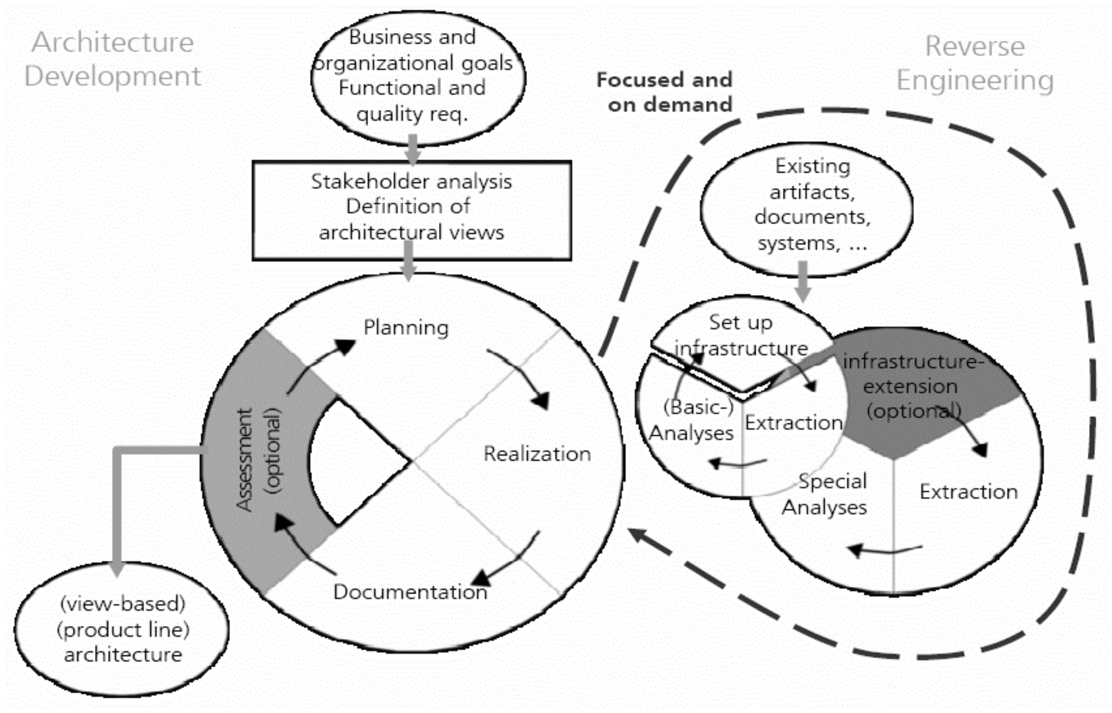

Architecture Development
Business and organizational goals Functional and quality req.
Stakeholder analysis Definition of architectural views
Planning
Realization
Documentation
Assessment (optional)
(view-based) (product line) architecture
Focused and on demand
Reverse Engineering
Existing artifacts, documents, systems, ...
Set up infrastructure
(Basic-) Analyses
Extraction
infrastructure-extension (optional)
Special Analyses
Extraction

Figure 7-1: Overview PuLSE-DSSA and ADORE

ADORE™ (Architecture- and Domain-Oriented Reengineering) is a request-driven reengineering approach that evaluates existing components with respect to their adequacy and, potentially, integrates such components into the product line:

- First, existing components are identified and reverse engineered [4] to assess their adequacy; this activity is initiated by requests coming directly from the product line architects.
- Second, based on the analysis results, the product line architects decide whether the existing component is reused or a new product line component is developed from scratch.
- Finally, when reusing the component, necessary renovation and extension activities are kicked off to adapt the component for its use within the product line.

ADORE is mainly instantiated in step 2 of PuLSE-DSSA (realization), when the architects reason about whether or not to reuse existing components. The architectural needs drive the selection of appropriate reverse engineering analyses. Analyses and, potentially, renovation activities are conducted asynchronously to the PuLSE-DSSA iterations. That is, the current iteration of the architecture development may proceed even if the ADORE activities are delayed. The advantage of such a demand-driven approach is that investment is kept as small as possible: only reverse engineering activities are performed, renovations are conducted only after the decision to include the component in the product line.

To enable stakeholder reasoning about such a decision to be made, certain aspects of the component have to be lifted to a higher level of abstraction. Existing component artifacts (e.g., source code, documentation, configuration files) are therefore exploited and the information extracted is aggregated in a repository. Since the repository usually has a lot of content, relevant information is often hidden in overcrowded low-level models. Thus, further analysis activities process the information and aim at creating meaningful views on the existing component.

Typical goals of the reverse engineering part in ADORE address the evaluation of the internal quality of a component, the degree of variability support, the provided flexibility towards anticipated changes, the compliance of a component to the product line architecture, or in case there are several similar implementations of a component, the identification of commonalities among them.

7.4 Analysis of Component Adequacy

An existing implementation of the Graphics component was identified (written in C++, approximately 180 KLOC) in the domain of the multimedia system. At the time of the analysis, the component had to be adapted to deal with a new hardware technology, so the source code was not yet fully available due to this technology change. The product line architects were doubtful whether the existing Graphics component was adequate for the product line and suitable to the architecture designed with PuLSE-DSSA. Therefore, we instantiated the ADORE approach and reverse engineered the Graphics component to the answer the following questions:

- Was the component implemented accordingly to its documentation, how consistent is the documentation and can it integrated seamlessly into the product line architecture? To answer these questions we applied **static architecture evaluations**.
- To which degree contains the subject component already existing variability? Is it possible to relate these code-level variations to higher levels (in best case to the product map coming from scoping activities)? To address this request, we conducted a **variability analysis** and refactored prototypically some variability by means of a frame processor.
- What are maintenance risks of the current implementation? This request triggered a set of reverse engineering activities: source code analysis including **clone detection**, the **metric computation**, a **naming and decomposition analysis**.
- Another request of the architects was concerned with the potential evolution of the algorithms and implementation decision made so far. We conducted a review of **code comments** to address this aspect.

7.4.1 Static Architecture Evaluation

The consistency of the component to its documentation was statically evaluated with the help of the SAVE tool (see [9], based on the idea of Reflexion models [10]). The component engineering models decomposed the subject into the three internal layers and provided a mapping to the source code files. Figure 7-2 depicts the results of the evaluation. The evaluation shows a high degree of consistency so far since there are almost no violations to the documented component engineering model (Layer-1 uses Layer-2, grey arrow, cardinality 1149); there are only two exceptions: the divergences form Layer-2 to Layer-1 (blue arrow, cardinality 2) and the absence from Layer-2 to Layer-3). The reason for the latter is that the component is currently still under development (the layer was only realized in stubs). The evaluation showed that the implementation so far did follow the intended design decisions, although detailed analysis of the layers gave pointers for improvement.

The challenge for the development organization is now to ensure this over time. The component's evolution has to be monitored when new variants are created based on this first product line component. To keep the quality and to avoid degeneration, we recommended quality assurance activities including the repetition of static architectural evaluations at defined checkpoints.

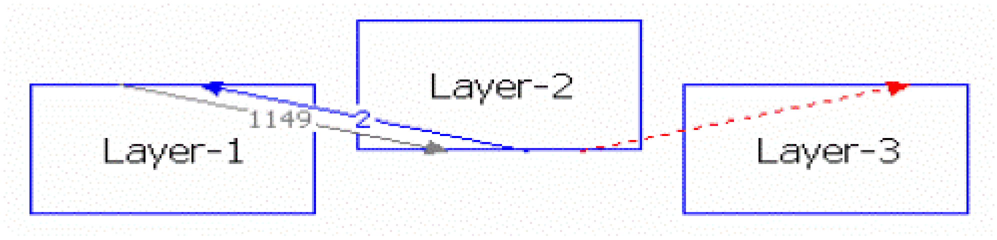


Figure 7-2: Component Internal Layers

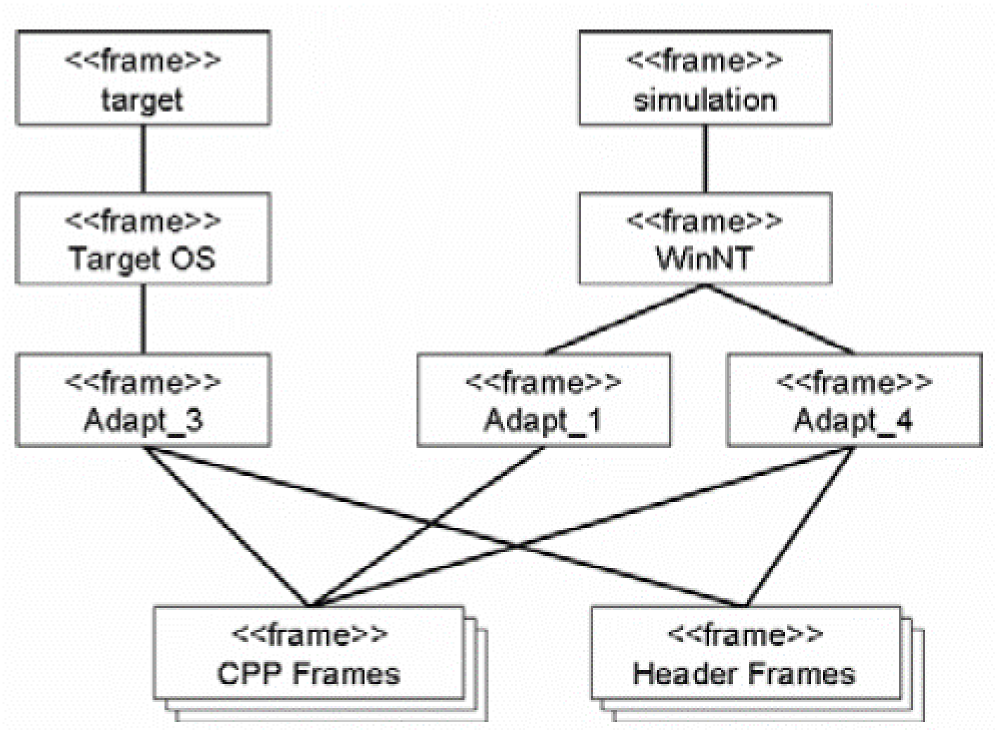


Figure 7-3: Frame Hierarchy

7.4.2 Variability Analysis

Conditional compilation with macros is a common means to realize variants in the source code. Optional or variable code parts or alternative implementations can be implemented in a common source code base (with preprocessor commands #if, #ifdef, etc.), and the resolution of the variants is taken over by the preprocessor. The variability analysis checked to which extent the macros and compile switches realize variability with respect to the product map. Identified variability was documented to make the variation points explicit and derive a decision model that relates the macros to the different members of the product line.

Furthermore, we exemplified how to extract and to migrate the current variabilities into more advanced tools like frame processors. A frame processor is a tool supporting frame technology [1], a technique to support reuse in practice. In frame technology, the implementation units, called frames, have the same appearance as those in any major programming language. They form a group of symbols (e.g., source code or frame code) that can be consistently referenced. Frames contain both source code and frame-specific code providing adaptation, which enables reuse. Frame-specific code consists of frame commands and frame variables in order to make variation points explicit by distinguishing between common and variable text. Frames can be arranged in hierarchies and will be resolved at compile time by the frame processor, an advanced preprocessor.

The frame processor processes the frame hierarchy and generates finally pure source code. In product family engineering, this technique is used to produce different product instances from a family by explicit variation points in one common code base. Figure 7-3 depicts the frame hierarchy operating system dependent thread handling for two system variants: the target variant and a simulation variant running on Windows. Frames positioned high in the frame hierarchy can adapt lower frames (by an ADAPT statement in the frame), on the lowest level there are the frames containing the commonalities among both variants (simulation and target), and they have explicit variation points. These variation points are adapted by higher level frames, for instance a VP filename.cpp_1 is replaced in the adapt_1.frame with “#include Windows.h”. The frame hierarchy was extracted automatically from the source code (leading to non meaningful names for the variation points and the lower level “adapt_*” frames).

The frame processor enables the explicit management of the frames, the hierarchy support and the automatic resolution of the variants. Adding a new variant involves only the creation of the respective frames. In the example, another OS variant would lead to the respective OS frame and an additional adapt frame. All variability and resolution is localized in these frames, and the developers work only with the two frames, and they have not to modify several files widespread in the implementation.

7.4.3 Clone Detection

A code clone is a code fragment that occurs in more than one place. One duplicate is usually the master, and the other one (i.e., the cloned one) is produced by copying the master (sometimes containing minor modifications). Code clones are a threat to the evolution code size, higher effort for maintenance (when there is the need for a change, all duplicates have to be addressed), reduced code readability, increased risk because an error can be propagated to several places in the source code which leads to high effort for the removal of such an error.

We analyzed the source code with a clone detection tool based on text pattern matching for two objectives. First, we aimed at detecting internal clones (duplicated code lines found in a single file), and second external clones (clones found in different files). The analysis identified a number of code clones with a size greater than 20 lines to be reviewed by the developers).

7.4.4 Metric Hotspots

Source code metrics are an objective means to learn about potentially problematic areas in the source code. By measuring coupling, size and complexity metrics and analyzing the outliers, unanticipated values, problematic areas and hotspots in the source code can be identified. In particular, we had significant outliers for the following metrics: cyclomatic complexity (for methods and class averages), CBO (coupling between object classes), NOC (number of derived children), and function size in terms of LOC (lines of code). The identified source code items have been triggered for code reviews, since such elements are error-prone, bring along the risk of unwanted side effects, and are difficult to understand. To avoid potential maintenance problems, these items are reviewed carefully.

7.4.5 Naming and Decomposition Analysis

When conducting a detailed source code analysis, a number of issues arose that became additional action items:

- File system representation: the folder structure and the code files did not reflect the decomposition as it was documented.
- Empty files: a couple of empty (or almost empty) files were identified (less than 20 LOC). The files are reviewed whether it is reasonable to merge them with other files or they can be eliminated from the code.
- Inconsistent naming conventions: although there were naming conventions in the code, they were not used consistently throughout the component.

7.4.6 Code Comments

A major issue was the ratio of commented lines to source code lines, which was below 10 percent. The developers agreed on improving this to facilitate the reading of the source code and to not run into problems when evolving the components further.

7.5 Summary

The reverse engineering results revealed that the Graphics component has a sufficient adequacy for the emerging product line and the product line architects decided to reuse the existing component. However, the results made the need for renovations and extensions obvious to fully address the product line requirements. An action list (improvement of the internal quality, assurance of consistency to the documentation and the intended design, refactoring of variabilities, removal of architectural mismatches, and the integration of the component) and items for a detailed analysis (code reviews and inspections) were derived directly from the reverse engineering results. For the final acceptance of the Graphics component as a product line component, these issues should be revisited.

The well-invested effort for reverse engineering lead to an architectural reuse decision that was well-grounded and sound, based on the reverse engineering results. Since the component was not yet fully implemented, the developers were able to address most of the suggested renovation items promptly in the ongoing development.

In summary, this experience report presents a typical industrial case where Fraunhofer PuLSE™ and ADORE™ are applied in combination. As it was in this case, there is generally a need for adaptation when components developed for single systems should become part of a product line infrastructure. In our experience so far, as-is reuse mostly does not work since there is always a need for adaptation to make the existing component suitable for the product line.

Hence, there is a strong need for efficient and focused reverse engineering analyses that support the reuse decision making, in this case by analyzing the adequacy of the existing components. In addition, it is important to identify and estimate the degree of required adaptation base the product line architects reuse decision on a well-grounded fundament.

7.6 References

[1] P. G. Basset, *Framing Software Reuse: Lessons From The Real World*, Prentice-Hall, 1996.

[2] J. Bayer et al.: "PuLSE: A Methodology to Develop Software Product Lines," 5th Symposium on Software Reusability (SSR'99), 1999.

[3] J. Bayer et al: *Definition of Reference Architectures Based on Existing Systems*, (IESE-Report 034.04/E), 2004

[4] E. Chikofsky, J. H. Cross: Reverse Engineering and Design Recovery: a Taxonomy, *IEEE Software, 7*(1):13-17, January 1990.

[5] P. Clements, R. Kazman, M. Klein: *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

[6] P. Clements, L. M. Northrop: *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[7] C. Hofmeister, R. Nord, R., D. Soni: *Applied Software Architecture*. Addison-Wesley, 1999.

[8] P. Kruchten: The 4+1 View Model of Architecture. *IEEE Software*, November 1995 12(6):42–50.

[9] P. Miodonski, T. Forster, J. Knodel, M. Lindvall, D. Muthig: *Evaluation of Software Architectures with Eclipse*, Kaiserslautern, 2004, (IESE-Report 107.04/E)

[10] G. C. Murphy, D. Notkin, K. Sullivan: Software reflexion models: bridging the gap between source and high-level models, *ACM Software Engineering Notes*, 1995.

8 Mining Existing Software Product Line Artifacts using Polymorphic Dependency Relations[1]

Igor Ivkovic and **Kostas Kontogiannis**
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L3G1 Canada
{iivkovic, kostas}@swen.uwaterloo.ca

Abstract

Development of a product line architecture involves mining existing software assets, from architecture-level design knowledge to implementation-level artifacts. Each mining effort is generally associated with an appropriate mining context, through which the criteria for component identification and selection are defined. The crux of the matter is variability, where a mining context has to be specific, to allow for precise component querying, but it also has to be adaptable and extensible, to accommodate the needs of different software product line instances. In this paper, we introduce a framework for annotating and querying heterogeneous software artifacts using polymorphic dependency relations in software product line reengineering. The dependency relations are defined based on the theory of semantic values, where an association rule is represented as a combination of different semantic properties and values (semantic contexts), such as features and constraints defined at the model or meta-model level. The chosen association rules denoted through a mining context are used to query individual component annotations.

Keywords: software reengineering, software product lines, mining existing assets, semantic value theory, polymorphic dependency relations

8.1 Introduction

In model-driven software evolution, software artifact models are changed at different levels of abstraction. For instance, use case models are used to apply change at the requirements specification level while deployment diagrams are used to manifest change at the deployment and integration levels. A mutation of an artifact model at one level may affect models at the same or at different levels of abstraction and detail. For example, a change in a design model could directly affect architectural models at the higher-level, and implementation models at the lower-level of abstraction.

To enable impact analysis and propagation of changes that may arise due to evolu-

1 This work is funded in part by the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

tion, it is necessary to establish and maintain associations among related models and their elements. In previous research [3, 4], we have introduced an approach for the identification and encoding of dependency relations among heterogeneous software artifacts using formal concept analysis (FCA). As part of the approach, each software model that is MOF compliant [6] is represented in terms of its objects and attributes. Objects that share common attributes are considered dependent, and are identified using a FCA algorithm [2]. To match attributes of heterogeneous contexts, we have introduced the notion *attribute association rules*. At the domain model or metamodel levels, attribute associations represent the functions for mapping compatible types and relations, while at the model level, they are used to map attributes, features, and annotations of individual model elements. The approach was applied to establish dependency relations between business process models and enacting Java source code by mining information flow models using business workflow patterns.

In this paper, we extend our FCA-based approach by defining dependency relations using the theory of semantic values [8]. Each dependency relation represents a composition of individual semantic values. The meaning of a relation is indicated by an association context, which represents a set of semantic properties and values. For example, semantic values Class(Language='UML'(Represents='Objects')) and class(Language='Java'(Represents='Objects')) are associated based on the context {Represents='Objects'}, but in contrast, are not associated based on the context {Language='UML'}. We treat each association context as a polymorphic type, for which we can derive a subtype by extending the association context (i.e., reduce its scope), or a supertype by reducing the association context (i.e., extend its scope).

In the context of software product line reengineering, the polymorphic types are first used to annotate existing software assets. Then, a mining context is defined as a combination of different association contexts, for example, with different contexts for different types of artifacts. Using the mining context as basis, candidate components are mined, and the most suitable selected for reuse in product line development.

The remaining content of this paper is organized as follows: Section 8.2 explains semantic annotations of existing assets using semantic heads. Section 8.3 describes the structure of the mining context and explains how the mining context is used to query candidate components. Finally, Section 8.4 provides our conclusions and directions for future research.

8.2 Semantic Annotation of Software Artifacts

The Options Analysis for Reengineering (OAR) approach [1] prescribes that after creating a mining context, it is necessary to inventorize available components, and identify their functionality, language, infrastructure support, and interfaces. Based on this inventory, candidate components can be selected as the ones that match the criteria of the mining context. However, the OAR description does not provide a formalism for specification of component properties, nor does it provide an algorithm for matching the criteria of the mining context and individual components.

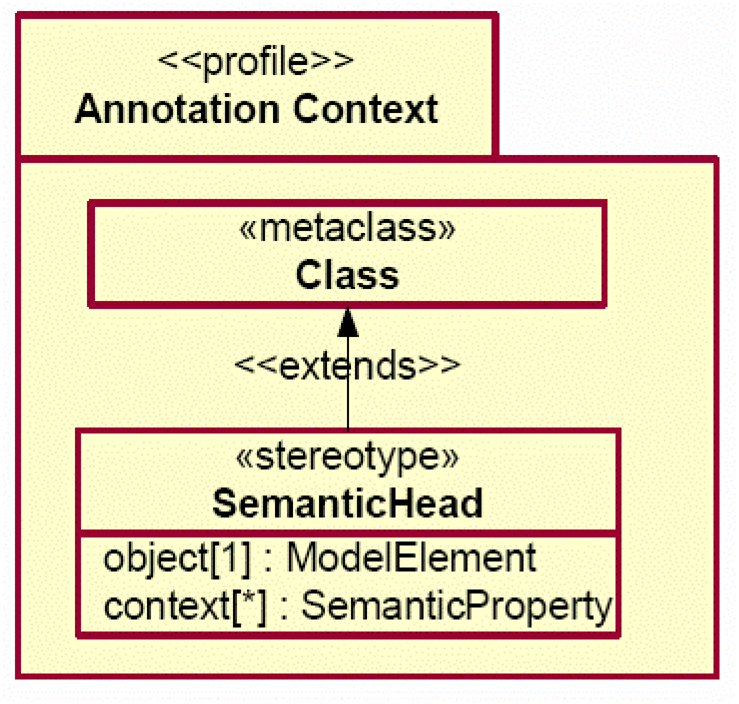


Figure 8-1: Annotation Context UML Profile

We propose to define a systematic approach to annotation of existing software assets by associating each available component with a corresponding semantic head. Each semantic head represents a set of semantic values, defined according to the theory of semantic values as described above [8]. As part of our view of software artifacts as MOF-compliant models, we use UML 2.0 metamodel [7] as the basis for representation. Hence, as shown in Figure 8-1, we define <<semanticHead>> stereotype as part of the Annotation Context UML profile. Each semantic head is associated with one model element, and it contains zero or more semantic values. The elements of the semantic head are created as part of the component inventory, and they may include implemented features such as (Feature=‘DatabaseAccess’, Limitation=‘DataManipulation’), interface properties such as (InterfaceType=‘Proxy’(Protocol=‘HTTP’)), language properties such as (ImplementationLanguage=‘ Java’(Dialect=‘Enterprise Java Beans’)), and environment constraints such as (PlatformIndependence=‘ Yes’(OperatingSystem=‘Windows’)).

8.3 Defining the Mining Context

Once we have annotated components with specific semantic properties, we can query them to identify those of specific interest and suitability for reuse in reengineering towards product lines.

As shown in Figure 8-2, we create the mining context as the collection of specific association rules. We represent each rule as a collection of semantic properties and values that are used to query individual component annotations. For instance, to find all components that use HTTP protocol for communication, we could use {Protocol='HTTP'} as the association context. We can also have different association contexts for different component types, for example, for areas such as database access, role-based access control, and user interfaces.

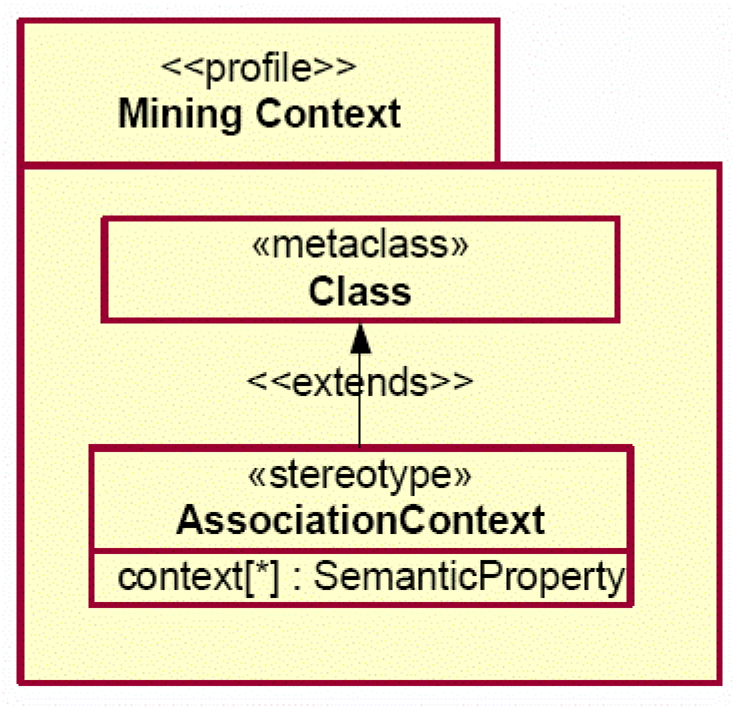
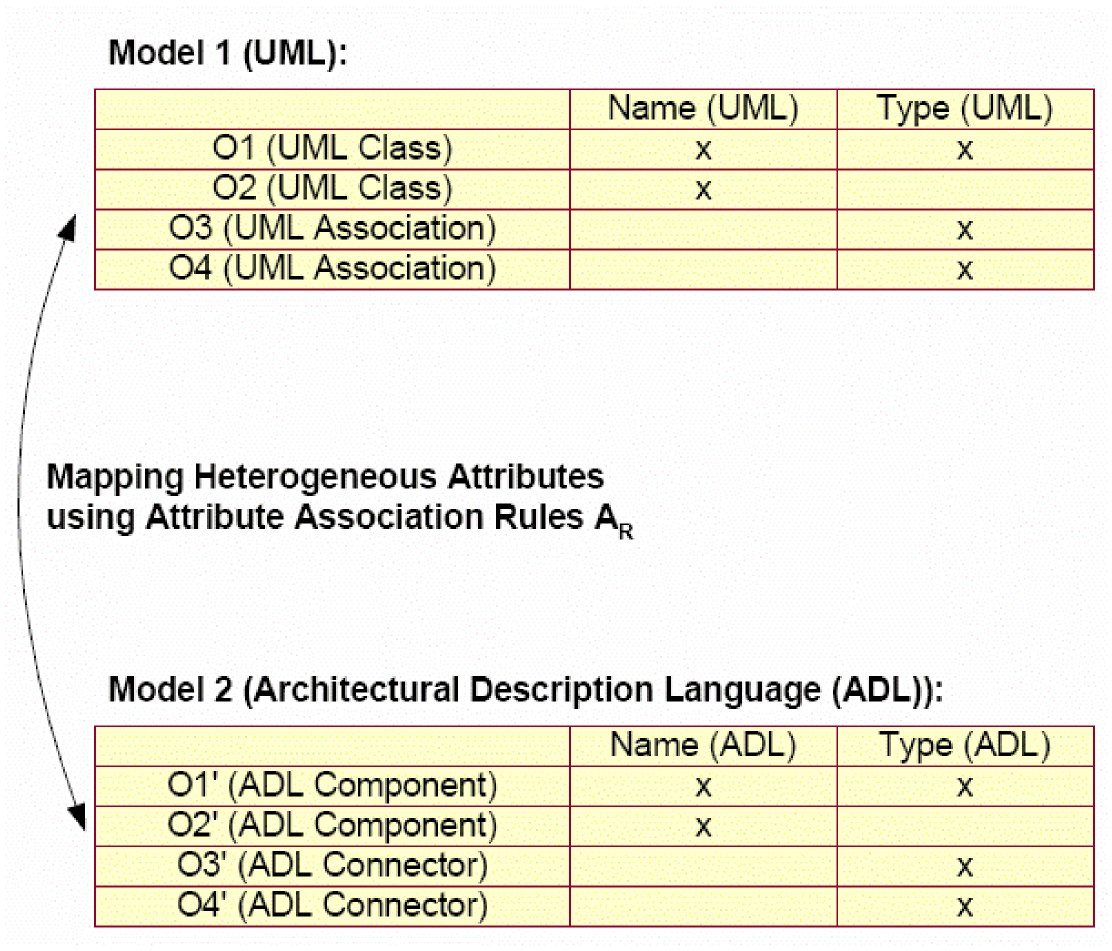


Figure 8-2: Mining Context as a UML Profile



Model 1 (UML):

	Name (UML)	Type (UML)
O1 (UML Class)	x	x
O2 (UML Class)	x	
O3 (UML Association)		x
O4 (UML Association)		x

Mapping Heterogeneous Attributes using Attribute Association Rules A_R

Model 2 (Architectural Description Language (ADL)):

	Name (ADL)	Type (ADL)
O1' (ADL Component)	x	x
O2' (ADL Component)	x	
O3' (ADL Connector)		x
O4' (ADL Connector)		x

Figure 8-3: Attribute Association Rules for Mapping Heterogeneous Semantic Values

For compatible properties that are at different levels of granularity or scale, conversion functions may be used. For instance, contexts (ImplementationLanguage=‘Java’) and (ImplementationLanguage=‘ObjectOriented’) can be mapped using cvtImplementationLanguage to represent Java at a higher level of granularity as an object-oriented language. Also, semantic value 30(Metric=‘ AccessPerformance’, MetricScale=‘miliseconds’) can be converted into 0.3(Metric=‘AccessPerformance’, MetricScale=‘ seconds’) by using the scaling function cvtMetricScale with the scaling factor 100.

For incompatible properties, such as properties from different domains as shown in Figure 8-3, conversion may be performed using attribute association rules including:

- Feature hierarchies, where contexts are matched if they related to a selected feature or one of its subfeatures.
- Lexicographical matching, where contexts are matched as text using various information retrieval techniques such

as n-gram matching, word-matrix matching, or latent-semantic indexing.

- Spatial matching, where contexts are matched based on their relation to a specific data flow.

8.4 Conclusions and Future Research

In this paper, we have presented a framework for defining the mining context for mining existing software assets using the theory of semantic values. We have presented an approach for annotating available components with corresponding semantic properties and values. We have also discussed the creation of the mining context using association rules, and use of the defined association rules to query and select suitable components.

In future research, we aim to extend the approach by more formally specifying the annotations and annotation categories. We also intend to adapt the approach to other mining steps as described in the OAR approach, such as component refactoring, and relate it to more recent reengineering methods such as the Service-Oriented Migration and Reuse Technique (SMART) [5].

8.5 Acknowledgements

This work is performed in collaboration with the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

8.6 References

[1] J. Bergey, D. Smith, N. Weiderman, and S. Woods. *Options analysis for reengineering (oar): Issues and conceptual approach*. Technical Report CMU/SEI-99-TN-014, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1999. http://www.sei.cmu.edu /publications/documents/99.reports /99tn014/99tn014abstract.html

[2] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[3] I. Ivkovic and K. Kontogiannis. Towards automatic establishment of model dependencies. *Accepted to the International* Journal of Software Engineering and Knowledge Engineering *(IJSEKE)*, Sep 2005.

[4] I. Ivkovic and K. Kontogiannis. Using formal concept analysis to establish model dependencies. In *Proceedings of the* IEEE International Conference on Information Technology *Coding and Computing*, Las Vegas, NV, Apr 2005.

[5] G. Lewis, E. Morris, L. O'Brien, D. Smith, and L. Wrage. *Smart: The service-oriented migration and reuse technique*. Technical Report CMU/SEI-2005-TN-029, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Sep 2005. http://www.sei.cmu.edu/publications /documents/05.reports/05tn029.html

[6] OMG. Meta object facility (mof) specification version 1.4. Technical report, Object Management Group (OMG), Apr 2002. http://www.omg.org/docs /formal/02-04-03.pdf.

[7] OMG. Unified modeling language (uml) 2.0 infrastructure specification. Technical report, Object Management Group, Sep 2003. http://www.omg.org/docs/formal /03-09-15.pdf.

[8] E. Sciore, M. Siegel, and A. Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems*, 19(2), June 1994.

9 Workshop Outcomes

The results of this workshop consist of the papers presented during it (included in Sections 4-8), the outcomes of discussions triggered by those papers, and some agreements related to the organization and continuation of this workshop and research topic.

The papers that were presented could roughly be said to address two main issues:

1. variation points (papers by Olumofin and Cornelissen)
2. identification of product line assets (papers by Ganesan, Knodel, and Ivkovic)

Part of the research is focused on the application and extension of existing reverse and reengineering techniques to a product line context. The problem that needs to be solved is how to use these techniques (that were previously applied only to individual systems) to handle multiple software variants.

For instance, in the case of dynamic analysis, the use of reverse and reengineering techniques implies additional difficulties with respect to the determination of useful scenarios to obtain traces from different variants of a software system. Cornelissen used this approach for the detection of potential variation points in a product line architecture.

Interestingly, the technique presented by Olumofin requires exactly that information about variation points. When combined with knowledge on sensitivity points (which can be discovered using the SEI Architecture Tradeoff Analysis Method® [ATAM®] developed by the Carnegie Mellon® Software Engineering Institute [SEI]) evolvability points can be identified that deserve special attention during software evolution to ensure the architectural conformance of the product line architecture and product family members.

Most approaches, however, focused on the use of static information. Two approaches were presented to find the software components that should be considered reusable assets for a product line. The approach by Ganesan and Knodel is based on metrics, while the approach proposed by Ivkovic uses semantic annotations to find the components in which there is interest. Finally an approach was presented that combines several techniques, such as metrics and clone detection, to assess the extent to which an existing software component is suitable for reuse in a product line environment.

Various problems involving the introduction of software product lines in software development organizations were covered in this workshop. The focus was on the

- detection of variability and how to use this information to ensure successful software evolution

® Architecture Tradeoff Analysis Method, ATAM, and Carnegie Mellon are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

- identification of components that are amenable to become product line assets
- determination of the extent to which these components are suited to be product line assets as is

Other issues that still need to be investigated in the future include

- the derivation of a complete product line architecture (instead of focusing on identifying individual components)
- combinations of different approaches, such as dynamic analysis and metrics or other static approaches
- the process steps involved in the migration towards a product line approach
- the scalability of the proposed approaches and potential tool support for them, as product line architectures typically concern large systems
- the reusability of other artifacts after migration to a product line approach, such as test cases and architectural views
- traceability from legacy to product line artifacts

One more important issue came up: participants found it difficult to find suitable case studies for applying their techniques. Such case studies should consider not only the availability of a complete product line example system but also a set of existing software variants that can be migrated to a software product line.

As a follow-up of this workshop, a mailing list was set up (r2pl@st.ewi.tudelft.nl) and the need for a successor workshop in 2006 was confirmed.

References

URLs are valid as of the publication date of this document.

[Graaf 05]	Graaf, Bas; O'Brien, Liam; & Capilla, Rafael. "Reengineering Towards Product Lines (R2PL2005)," 231. *Proceedings of WCRE 2005, 12th Working Conference on Reverse Engineering*. Pittsburgh, PA, USA, November 10, 2005. Los Angeles, CA: IEEE Computer Society, 2005.
[R2PL 05]	*Workshop on Reengineering Towards Product Lines (R2PL 2005).* http://www.st.ewi.tudelft.nl/~basgraaf/r2pl2005 (November 2005).

REPORT DOCUMENTATION PAGE				*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 2006			3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE R2PL 2005—Proceedings of the First International Workshop on Reengineering towards Product Lines				5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) Bas Graaf, Liam O'Brien, Rafael Capilla				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213				8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2006-SR-002
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116				10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS				12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) This report contains the proceedings from the First International Workshop on Reengineering Towards Product Lines (R2PL) 2005, which was held on November 10th, 2005 in Pittsburgh, Pennsylvania, USA and colocated with the Working Conference on Reverse Engineering (WCRE) 2005 and WICSA 2005—the Working Institute of Electrical and Electronics Engineers/International Federation for Information Processing (IEEE/IFIP) Conference on Software Architecture. This report consists of an overview of an invited presentation, a set of position papers, and details of the workshop's outcomes.				
14. SUBJECT TERMS software product lines, reverse engineering, reengineering				15. NUMBER OF PAGES 72
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified		20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102